Nonatomic dual bakery algorithm with bounded tokens

Aravind, Alex A.; Hesselink, Wim H.

Published in:
Acta informatica

DOI:
10.1007/s00236-011-0132-0

Publication date:
2011

Acta Informatica (2011) 48:67–96
DOI 10.1007/s00236-011-0132-0

ORIGINAL ARTICLE

Nonatomic dual bakery algorithm with bounded tokens

Alex A. Aravind · Wim H. Hesselink

Received: 6 April 2010 / Accepted: 20 January 2011 / Published online: 16 February 2011

Abstract A simple mutual exclusion algorithm is presented that only uses nonatomic shared variables of bounded size, and that satisfies bounded overtaking. When the shared variables behave atomically, it has the first-come-first-served property (FCFS). Nonatomic access makes information vulnerable. The effects of this can be mitigated by minimizing the information and by spreading it over more variables. The design approach adopted here begins with such mitigating efforts. These resulted in an algorithm with a proof of correctness, first for atomic variables. This proof is then used as a blueprint for the simultaneous development of the algorithm for nonatomic variables and its proof. Mutual exclusion is proved by means of invariants. Bounded overtaking and liveness under weak fairness are proved with invariants and variant functions. Liveness under weak fairness is formalized and proved in a set-theoretic version of temporal logic. All these assertions are verified with the proof assistant PVS. We heavily rely on the possibility offered by a proof assistant like PVS to reuse proofs developed for one context in a different context.

1 Introduction

The problem that concurrent processes may need exclusive access to some shared resource, was first proposed and solved in [7]. The problem came to be known as mutual exclusion in [8]. Numerous solutions to this problem have been proposed. Surveys can be found in [4,30,34]. Among the proposed algorithms, Lamport's bakery algorithm [20] and Peterson's algorithm [28] deserve special mention for their simplicity and elegance.

A. A. Aravind (✉)
Computer Science Program, University of Northern British Columbia,
Prince George, BC V2N4Z9, Canada
e-mail: csalex@unbc.ca

W. H. Hesselink
Department of Computing Science, University of Groningen, P.O. Box 407,
9700 AK Groningen, The Netherlands
e-mail: w.h.hesselink@rug.nl

In particular, Lamport's bakery algorithm has the nice properties that competing processes are served in first-come-first-served order (FCFS) and that the shared variables used need not be atomic [20]. The latter property means that the algorithm does not assume mutual exclusion on read and write operations on shared variables (this is known as the noncircularity property). The downside is that these shared variables hold integer tokens that can become arbitrarily large. Several modifications to the bakery algorithm have been proposed to bound these tokens [2,15,16,32,33,35,38]. All these modifications, however, require some atomic shared variables. Our algorithm bounds the tokens and works for safe (nonatomic) shared variables.

The proof of correctness of a concurrent algorithm, especially when it uses nonatomic shared variables, always requires many case distinctions. In our approach, we need to go over the proof several times under subtly changed conditions. This can be done much more reliably by a mechanical proof assistant than by a human verifier. We used the proof assistant PVS [27] for this purpose.

1.1 Relevance of nonatomic access

Nonatomic algorithms are practically relevant now, because several recent systems such as smart-phones, multi-mode handsets, network processors, graphics chips, and other high performance electronic devices use multiport memories, and such memories allow nonatomic accesses through multiple ports [14,31,39]. Such developments were foreseen long ago, e.g., in [29, p. 58] with a long paragraph explaining the application to VLSI chips with thousands of processors (becoming reality now).

Indeed, shared memory with atomic access is an abstraction that is extremely useful in the design of all kinds of concurrent or distributed systems, but atomic algorithms require arbitration on a lower level [18,23]. Our algorithm truly solves the mutual exclusion problem directly.

Safeness is the weakest well-formalized weakening of atomicity that we are aware of. A shared variable is called *safe* [22] if, in a system where it is written by only one thread at a time (and therefore in some sequential order), every read operation that does not overlap with any write operation returns the most recently written value, and every read operation that does overlap with some write operation returns an arbitrary value of correct type.

There is an extensive literature (e.g. [36]) on the wait-free implementation of atomic variables by means of safe variables. Indeed, Haldar and Subramanian [9,11] construct an atomic shared variable for one writing process and one reading process by means of safe variables. Based on this, one can use, e.g., the algorithm of Vitányi and Awerbuch [37] to construct atomic variables for several reading and writing processes, on top of which any mutual exclusion algorithm for atomic variables can be applied. This would yield a layered algorithm for mutual exclusion for bounded safe variables. Based on [37], such a construction requires many safe copies for each atomic shared variable, and every read and write action at the atomic level requires many reads or writes of the lower level. We therefore expect a layered solution to need significantly more space and time than the corresponding atomic one.

1.2 Is it relevant to bound the integers?

If one works with 32 bit integers on a 3 GHertz machine, Lamport's bakery algorithm might reach overflow within a minute, i.e., the occurrence of integers that cannot be represented on the machine. Then errors may occur. If overflow is ignored, mutual exclusion can be

violated. Otherwise unexpected exceptions may be raised. The occurrence of overflow may be extremely unlikely, but algorithms for mutual exclusion are typically used in operating systems or embedded systems that must function reliably during extended periods while the context in which errors might occur is unknown. Then one cannot afford errors, even with very small probabilities [10, p. 33]. In our view, therefore, bounding the integers is very important for the applicability of the algorithm.

Mutual exclusion without FCFS does not require much shared memory. Indeed, the paper of Peterson [29] gives a mutual exclusion algorithm for N threads with $2N$ *safe* Boolean variables. The paper mentions a solution with one safe variable per thread that can have 3 values. His solutions do not assure FCFS and have a logic more complex than ours. Anderson [3] gives a tournament algorithm based on Kessels' solution [18] for 2 threads, but this version also does not assure FCFS.

Mutual exclusion with FCFS requires that the order of the threads in the waiting queue is encoded in memory. Therefore, the number of different values of the complete state must be at least $N!$ where N is the number of threads. By Stirling's formula, $N!$ is approximated by $\sqrt{2\pi N}N^N e^{-N}$. This indicates that it is reasonable to use N shared variables that can each have N different values. It is also possible, however, to encode the order of the waiting queue partly in private variables.

In terms of shared-space complexity, the best mutual exclusion algorithm with FCFS known is the one of Lycklama and Hadzilacos [25] that uses 5 safe shared bits per thread, and thus a shared state space of 2^{5N} elements which is asymptotically much smaller than $N!$. This algorithm therefore encodes part of the order of the waiting queue in private variables. It works by a beautiful separation of concerns in which mutual exclusion and FCFS are treated by different parts of the algorithm.

Our algorithm in [6] uses N *write-safe* variables (see Sect. 1.6 for the definition) with values in $[0 \ldots N-1]$, and N safe shared bits. Taubenfeld's algorithm [33] uses N atomic variables with values in $[0 \ldots N]$, and $N+1$ atomic shared bits.

The algorithm proposed here uses N safe variables with values in $[0 \ldots N]$, and $3N+2$ safe shared bits. To summarize, it is relatively simple, solves the mutual exclusion problem directly, bounds the tokens, and satisfies bounded overtaking (see Sect. 1.4) and quasi-FCFS (i.e. FCFS with an unbounded doorway, see Sect. 4.2).

1.3 General principles

A read of a safe variable can return any value that the variable can hold. We see no general abstractions underlying our approach, but we do have the following rule-of-thumb principle for the design of algorithms for safe variables.

If one has to rely on safe shared variables, one should minimize the vulnerable information and spread it over several variables. The reason for this is that reading a safe variable gives only information about the value of the variable when the writing thread is not executing a write instruction. The information obtained by the reading thread is therefore the value intended or the fact that the writer is executing a write instruction, but it does not know which of the two cases applies. Reading different safe variables one after the other gives therefore more information than reading them together as a safe structured variable that holds them all.

More concretely, for example, in the `max` based token computation of the bakery and black-white bakery algorithms, a read of the token can return any value that the token can hold. One of the ways to minimize the effect of this nondeterminacy is to reduce the number of possible values that a read can return. In our count based token computation, a read just returns a Boolean value.

Our approach is thus as follows. We first minimize and spread the vulnerable information. Then we prove correctness for the case with atomic shared variables using invariants. The assumption of atomicity is then removed in steps while retaining the correctness proof. This may require weakening of the invariants, or the introduction of additional waiting. The approach has no success guarantee. Successful application requires a proof assistant like PVS, because one needs to repeatedly prove invariants under similar but subtly modified conditions. A mechanical tool is better for such a task than an intelligent human being.

1.4 Problem setting

The setting is traditionally modeled as follows. There are N threads (or processes) that communicate via shared variables and that repeatedly compete for a shared resource. The threads are thus of the form:

thread $(i : 0 \leq i < N) =$
  loop
    NCS ; *Entry* ; *CS* ; *Exit*
  end.

Here, *NCS* and *CS* are given program fragments that stand for the noncritical section and the critical section, respectively. *NCS* need not terminate, *CS* is guaranteed to terminate. The problem is to implement *Entry* and *Exit* in such a way that the number of threads in *CS* is guaranteed to remain ≤ 1 (mutual exclusion). It is required that *Exit* is wait-free, i.e., every thread can pass *Exit* without waiting, in a bounded number of its own steps.

The progress requirement is that, whenever there are threads in *Entry*, eventually some thread will enter *CS*. Freedom from starvation is the condition that, if some thread has entered *Entry*, eventually it will enter *CS*.

The first-come-first-served property FCFS is defined as follows [20]. It is required that the program fragment *Entry* is a sequential composition of two fragments *Doorway* and *Waiting*, such that *Doorway* is wait-free and that, if a thread p has passed *Doorway* when a thread q has not yet entered *Doorway*, thread q is not admitted to *CS* before thread p.

The property of *bounded overtaking* means that there is an upper bound U, say, such that, when some thread j has passed the doorway (or a well-specified part of it), no other thread can reach *CS* more than U times before thread j does. In the case of our algorithm, we have the upper bound $U = 2$.

1.5 Overview

In Sect. 1.6, we discuss nonatomic shared variables and their formalization.

In the Sects. 2, 3, 4, we present three mutual exclusion algorithms with different levels of atomicity. It is a case of refinement of atomicity, not in a formal sense, but in the sense of reuse of ideas, concepts, and proofs.

In Sect. 2, we introduce the dual bakery algorithm with atomic variables as a variation of the bakery algorithms of Lamport [20] and Taubenfeld [33]. We minimize the vulnerable information and prepare the ground for spreading it over different variables. The algorithm is developed and proved carefully, because all points raised reappear with distracting details in the subsequent refinements. We need quite a number of invariants to prove mutual exclusion (MX) and first-come-first-served (FCFS) for this atomic version.

Section 3 contains the "half-atomic" version of the algorithm. We now actually spread the information over different variables, which are still accessed in atomic actions. This

introduces a "race condition", which is eliminated by adding another test in the Doorway. The half-atomic version with this addition again satisfies MX and FCFS.

In Sect. 4, we finally allow the shared variables to be safe instead of atomic. The nonatomicity of the assignments forces us to introduce two new waiting conditions. In this way, mutual exclusion is proved again. We show that FCFS can be violated, but that a weakened version of it holds. On the other hand, a strong form of bounded overtaking is proved.

Progress for the nonatomic algorithm is treated in Sect. 5. We analyse how the next thread to enter the critical section is selected. We next use the safety properties to formally prove that under the assumption of weak fairness (see Sect. 5.4 for the definition) every thread always eventually comes back to the noncritical section.

The proofs of MX and of (a weakening of) FCFS for the nonatomic versions are to a large extent careful variations of the proof for the atomic version. Each of these proofs is based on a number of invariants that have been verified [12] with the proof assistant PVS [27]. The invariants for the atomic case survive with minor modifications for the nonatomic case. The complete list of invariants for the nonatomic algorithm is given in the appendix.

In Sect. 6, we briefly discuss how we used the model checker Spin [13] and the proof assistant PVS [27]. Conclusions are drawn in Sect. 7.

1.6 Nonatomic shared variables

As stated earlier, the threads can communicate via shared variables that need not be atomic. We therefore have to recall the concepts of safeness and atomicity of shared variables.

A shared variable is called *atomic* if read and write operations on it behave as if they never overlap but always occur in some total order that refines the *precedence* order (an operation precedes another iff it terminates before the other starts). Safeness is a weaker property. A shared variable that is written by only one thread is called *safe* [22] if every read operation that does not overlap with any write operation returns the most recently written value, and every read operation that does overlap with some write operation returns an arbitrary value of correct type.

If the assumption that the variable is written by only one thread is guaranteed statically, the variable is called an *output variable* [6]. If it is implied by a mutual exclusion property of the program that is to be executed, we encounter a circularity problem: we need the properties of the variables to prove mutual exclusion for the specific program and we need mutual exclusion to guarantee the properties of the variables.

We therefore define a program (or algorithm) to be correct for safeness of a set V of variables if, under the assumption that the variables in V are *write-safe*, the program satisfies its specification and guarantees mutual exclusion on V: never two (or more) threads have simultaneous write-access to any variable in V.

Here, we use the term *write-safe* as defined in [6]. A variable is write-safe if concurrent reading and writing of it is allowed and satisfies the following clauses: (1) every write action takes effect atomically at some moment during the action, (2) a read action not concurrent with any write action returns the current value, (3) a read action concurrent with a write action returns an arbitrary value of the correct type. Note that "write-safe" is stronger than "safe".

A read action of a safe shared variable $\mathtt{x}$ into a private variable v is written as $v := \mathtt{x}$. It can be regarded as atomic because it does not influence the shared state. We need only reckon with the possibility that it overlaps with one or more write actions. In order to indicate that a read action during a write action can return an arbitrary value, we denote a write action of a private expression E into a safe shared variable $\mathtt{x}$ by

$$\text{(flickering) } \mathtt{x} := E. \tag{0}$$

In relational semantics like TLA [24], we model this by a predicate like

$$pc' = pc \ \vee \ (pc' = pc + 1 \ \wedge \ \mathtt{x}' = E).$$

The primes are used for the values of the variables after the step, *pc* stands for the program counter, and by convention all variables apart from x and *pc* are unchanged. In other words, command (0) is modelled as a repetition of arbitrary assignments to x that ends with the assignment of E to x. The value of x during the repetition is indeterminate. We assume the liveness condition that the repetition terminates.

2 Dual bakery, atomic version

In this section, we present a mutual exclusion algorithm for N threads that communicate via atomic shared variables. In Sect. 2.1, we introduce and modify the priority determination as inherited from the algorithms of Lamport and Taubenfeld, and prepare the spreading of vulnerable information. The algorithm is presented in Sect. 2.2. Section 2.3 is an aside on the interpretation and implementation of waiting loops because almost all synchronization in this paper relies on it. Section 2.4 gives the proof obligations for safety of the algorithm. Section 2.5 presents the proof of correctness for the atomic case, as a preparation for the other two cases.

2.1 Priority determination

Lamport's bakery algorithm [20] determines the priority of threads to enter the critical section by means of a shared array

var tk : **array** $[N]$ **of** $\mathbb{N}$.

In the entry protocol, each thread i determines its token $\mathtt{tk}[i]$ by means of

$$\mathtt{tk}[i] := 1 + \mathrm{Max}\{\mathtt{tk}[j] \mid j \neq i\},$$

while it resets its token by $\mathtt{tk}[i] := 0$ in the exit protocol.

In Lamport's bakery algorithm, the tokens $\mathtt{tk}[i]$ can become arbitrarily large. In order to keep them bounded, Taubenfeld [33] uses colored tokens in his Black-White bakery algorithm and takes the maximum of the tokens of the same color. This can be visualized, equivalently, as using two queues. Entering threads join the *waiting queue*, while the threads in the *serving queue* enter the critical section one by one. When the serving queue is empty, the first thread of the waiting queue enters and swaps the queues. The data structure for the queues consists of a shared Boolean and a Boolean array:

var wq : $\mathbb{B}$,
var q : **array** $[N]$ **of** $\mathbb{B}$.

Thread i is in the waiting queue iff $\mathtt{q}[i] = \mathtt{wq}$.

Our aim is to allow nonatomicity of the shared variables. This means that the algorithm must not depend too much on the vulnerable information of single shared variables. Therefore, in our dual bakery algorithm, instead of taking the maximum of the tokens, every entering thread takes as its token the number of active threads in its queue, according to the assignment

$$\mathtt{tk}[i] := 1 + \#\{j \mid \mathtt{q}[j] = \mathtt{q}[i] \ \wedge \ \mathtt{tk}[j] > 0\}. \tag{1}$$

Here $\#S$ stands for the number of elements of set S. It follows that $\mathtt{tk}[i]$ is bounded between 0 and N.

In Lamport's bakery algorithm, the fragment *Entry* consists of three parts: the *Doorway* for token computation, the *Synchronization* to ensure that competitors leave the doorway, and *Waiting* for competitors that have priority. The Doorway of thread i is guarded by a Boolean flag $\mathtt{inDo}[i]$, which is initially false.

$$\textbf{var } \mathtt{inDo} : \textbf{ array } [N] \textbf{ of } \mathbb{B}.$$

In Synchronization, thread i waits for each j until $\mathtt{inDo}[j] = \mathit{false}$. The remarkable thing is that, when it has observed $\mathtt{inDo}[j] = \mathit{false}$, thread i is allowed to proceed while thread j may set $\mathtt{inDo}[j] := \mathit{true}$. This point is of course inherited from Lamport's bakery algorithm. A Boolean flag used in this way will be called a *guardian*.

Priority is based on token values, augmented with thread identifiers for tie-breaking. We encode this by comparing the values of $\mathtt{tk}[j] \cdot N + j$. In Waiting, therefore, thread i takes priority over thread j according to the Boolean value of

$$\mathtt{tk}[j] = 0 \ \vee \ \mathtt{tk}[i] \cdot N + i < \mathtt{tk}[j] \cdot N + j.$$

When there are two queues, priority of thread i over thread j is defined by

$$\begin{array}{l} \mathtt{tk}[j] = 0 \\ \vee \ \mathtt{q}[i] = \mathtt{q}[j] \ \wedge \ \mathtt{tk}[i] \cdot N + i < \mathtt{tk}[j] \cdot N + j \\ \vee \ \mathtt{q}[i] \neq \mathtt{q}[j] = \mathtt{wq}. \end{array} \tag{2}$$

Here, the first alternative means that j is not competing, the second means that i and j are in the same queue but thread i has priority because of token values, the third means that thread i is in the serving queue while j is in the waiting queue.

Remark When the same token numbers are chosen within a queue, the algorithm breaks the tie using thread identifiers, just as in [20,33]. The scheme is biased in the sense that it always favors threads with smaller identifiers. Because there are two queues, we can remove this bias by replacing $\mathtt{tk}[i] \cdot N + i < \mathtt{tk}[j] \cdot N + j$ in the second line of (2) by $\mathtt{tk}[i] \cdot N + s \cdot i < \mathtt{tk}[j] \cdot N + s \cdot j$ where $s = (\mathtt{q}[i] \ ? \ +1 : -1)$. Then, for threads i with $\mathtt{q}[i] = \mathit{true}$, the tie is broken in favor of the threads with lower numbers, while for threads i with $\mathtt{q}[i] = \mathit{false}$, it is broken in favor of the higher numbers. Then the scheme is unbiased because the chances of getting *true* or *false* are equal. We do not pursue this idea below, because it clutters the formulas unnecessarily. □

Just as with Taubenfeld's Black-White bakery algorithm, we need a fourth constituent in which the queues are swapped. In view of our version (1) of the token computation, we must not allow a competing thread of the current waiting queue to exit before all threads in the doorway have completed the doorway. The swapping thread therefore also needs a kind of synchronization.

2.2 The atomic dual bakery algorithm

Following [26], we assume that we cannot access two shared variables in a single atomic statement. Therefore, an entering thread i first reads $\mathtt{wq}$ into a private variable $oq.i$ before assigning $oq.i$ to $\mathtt{q}[i]$. We use the convention that shared variables are in `typewriter` font, and that private variables are *slanted*. If v is a private variable, we write $v.i$ for the value of v of thread i, but we use v itself in the code for thread i.

The doorway code uses a private counter *ccnt* to count the number of competing threads in the same queue and a private set *liDo* to hold the threads yet to be considered. In the atomic version we assume that the two arrays tk[] and q[] are encoded in a single array tq[]. This can easily be done because the number of different values for the pairs $\mathtt{tq}[j] = (\mathtt{tk}[j], \mathtt{q}[j])$ is $2 \cdot (N+1)$. We thus declare

var tq : **array** $[N]$ **of** $(\mathbb{N}, \mathbb{B})$.

Formula (1) is thus represented by:

Doorway code for thread i:
11 inDo$[i] := true$;
12 $oq :=$ wq ;
13 tq$[i] := (0, oq)$; $liDo := Thread \setminus \{i\}$; $ccnt := 1$;
14 **for** $j \in liDo$ **do**
$(n, b) :=$ tq$[j]$;
if $n > 0 \ \wedge \ b = oq$ **then** $ccnt := ccnt + 1$ **fi**
od ;
17 tq$[i] := (ccnt, oq)$;
18 inDo$[i] := false$.

We number the atomic commands from 11, for the ease of recognition. Numbers 15 and 16 is left open because below we need to separate the inspections of tk$[j]$ and q$[j]$.

The sets *liDo.i* are introduced because in the proof we need to know which threads have been treated by thread i in its loop 14. Line 14 is an atomic command, but this only means that the body of the loop is an atomic command, not that the loop as a whole is done atomically. Indeed, line 14 is equivalent to the transition

14 **if** *empty*(*liDo*) **then goto** 17 **else**
extract some j **from** *liDo* ;
$(n, b) :=$ tq$[j]$;
if $n > 0 \ \wedge \ b = oq$ **then** $ccnt := ccnt + 1$ **fi** ;
goto 14
fi.

As announced, Synchronization serves to enable all threads in the doorway to pass the doorway (although new threads may enter it). Synchronization is at line 21. It uses a private set *liSy*, which is initialized in the last command before 21.

Synchronization code for thread i:
(18) $liSy := Thread \setminus \{i\}$;
21 **for** $j \in liSy$ **do await** $\neg$ inDo$[j]$ **od** ;

We do not use separate line numbers for the initialization of private variables like *liSy*, because our line numbers stand for atomic commands that have to be considered when proving invariants.

Waiting for priority uses a set-valued private variable *liPri* to hold the set of threads thread i is still waiting for, and a private variable *th* for the chosen thread. The token comparison of formula (2) of Sect. 2.1 is thus encoded in:

Waiting code for thread i:
(21) $liPri := Thread \setminus \{i\}$;
22 **while** $\neg$ *empty*(*liPri*) **do**

```
            choose some th ∈ liPri ;
            (n, b) := tq[th] ;
            if n = 0 ∨ b = oq then
                if n = 0 ∨ ccnt · N + i ≤ n · N + th
                then remove th from liPri fi
24          else
                if wq ≠ oq then remove th from liPri fi
            fi
        od.
```

We have separated the lines 22 and 24 in such a way that the pair $\mathtt{tq}[th]$ is inspected in 22, and that in line 24 (only) the shared variable $\mathtt{wq}$ is inspected. We keep line number 23 available for splitting the inspection of the pair $\mathtt{tq}[th]$. Note that $(ccnt, oq)$ is a private copy of $\mathtt{tq}[i]$. In accordance with formula (2), roughly speaking, thread th is not removed from $liPri.i$ (so that thread i keeps waiting) when

$$n > 0 \ \wedge \ ((b = oq \ \wedge \ n \cdot N + th \leq ccnt \cdot N + i) \ \vee \ b \neq oq = \mathtt{wq}).$$

The remainder of the code is performed under mutual exclusion. It contains the critical section, the code for swapping the queues, and the exit code. We call it the *central code*, so that we can refer to this part in the later refinements. In comparison with [33], the swapping code needs an extra waiting loop with a private set variable $liSw$ because of our more primitive token computation.

Central code for thread i:

```
26      if wq = oq then
27          wq := ¬ oq ; liSw := Thread \ {i} ;
30          for j ∈ liSw do await ¬ inDo[j] od
        fi ;
31      critical section ;
33      tq[i] := (0, oq).
```

The line numbers 28, 29, and 32 will be used in the later refinements. The correctness proof is postponed to Sect. 2.5.

2.3 Implementation of waiting loops

This subsection is an aside to indicate how the performance of an implementation of the algorithm might be improved. This depends very much on the platform one uses and on the type of processors involved. The remarks made here will not be used elsewhere in the paper.

In the lines 21 and 30, we have waiting loops of the form

$$\textbf{for } j \in li \textbf{ do await } \neg \mathtt{inDo}[j] \textbf{ od}. \tag{3}$$

A naïve implementation would consider the threads one after the other. If the fraction of threads that are in the guarded region from 12 to 18 is f, and the time spent in the guarded region is Δ, this would imply the loop to take $\frac{1}{2}\Delta f(N-1)$ time on average.

The nondeterminate choice of the order of the threads in the **for** loop, however, allows a more efficient implementation. It first removes as many of them as are not in their guarded region:

```
for j ∈ li with ¬ inDo[j] do remove j from li od ;
for j ∈ li do await ¬ inDo[j] od.
```

The first loop does not involve waiting and should reduce the number of elements of li to around fN. The second loop should therefore require at most $\frac{1}{2}\Delta(1 + f(fN - 1))$ time on average, but can be expected to take less time because the threads that have exited their guarded region are unlikely to enter again quickly.

While the waiting loop (3) thus allows the implementation to consider the threads j in arbitrary order, we do not rely on it for progress. In other words, for the proof of liveness, we allow the implementation to regard loop (3) as disabled whenever there is a thread $j \in li$ for which `inDo`$[j]$ holds. We come back to this in Sect. 5.1.

2.4 Proof obligations for safety

We need many invariants to prove the correctness of the algorithm. If m is a line number, we write j **at** m to denote that thread j is at line number m, i.e., that $pc.j = m$. In particular, thread j has not yet executed command m. We write j **in** $[m \ldots n]$ to express $m \leq pc.j \leq n$, and j **in** $[m \ldots]$ for $m \leq pc.j$, etc.

Mutual exclusion can be expressed by the condition that, whenever some thread is at 31, no other thread is at 31. In order to prove this, however, we need to strengthen it by requiring, as announced, that the complete central code is executed under mutual exclusion. This is expressed in the invariant

MX: $\quad j$ **in** $[26 \ldots] \ \wedge \ k$ **in** $[26 \ldots] \ \Rightarrow \ j = k.$

Here and in all invariants to come, we universally quantify over all free variables (here j and k).

The atomic dual bakery algorithm also satisfies the first-come-first-served property FCFS. In order to prove this, we need to register, for every entering thread, which competing threads have passed the Doorway. Recall that a ghost variable (or auxiliary variable [26]) is a variable that has no role in the algorithm but is useful in the proof. We here introduce the ghost variable

`predec` : **array**$[N]$ **of set of** *Thread*

with the intention that `predec`$[i]$ holds the predecessors that thread i must give priority to because they had passed the Doorway when i entered. These sets are therefore updated by thread i in the lines 11 and 33 in the form:

11 `inDo`$[i] :=$ *true* ; `predec`$[i] := \{j \mid j$ **in** $[21 \ldots]\}$;
33 `tq`$[i] := (0, oq)$;
for all j **do** `predec`$[j] :=$ `predec`$[j] \setminus \{i\}$ **od**.

The loop in line 33 can be regarded as part of the atomic statement because `predec` is a ghost variable. The proof obligation is that thread i is not admitted to the critical section before any of its predecessors, i.e., before `predec`$[i]$ is empty. This amounts to the invariant:

FCFS: $\quad j$ **in** $[26 \ldots] \ \Rightarrow$ `predec`$[j] = \emptyset$.

2.5 Correctness: the two queues

We split the correctness proof in three parts: first the alternation between the two queues, as governed by the private variables $oq.j$, then priority within the queues determined by the values of `tk`$[j]$, which suffices to prove mutual exclusion, and finally the FCFS property determined by `predec`$[j]$.

Mutual exclusion can only be violated when some thread is in its central code, say k **in** $[26 \ldots]$, and some other thread, say j, exits the loop of 22 and goes to 26. The proof

that this does not happen first shows that, in this critical situation, the threads j and k are in the same queue. The second ingredient of the proof is a comparison of the token values.

We have to investigate the threads that are counted by thread i in line 14. For this purpose we introduce set-valued private variables $est.i$ to hold the set of these threads (est stands for estimate). Then the variable $ccnt$ can be removed. For simplicity in the proof, we split the pairs $\mathtt{tq}[j] = (\mathtt{tk}[j], \mathtt{q}[j])$ again. The lines 13 . . . 17 are thus replaced by

```
13      q[i] := oq ; tk[i] := 0 ; liDo := Thread \ {i} ; est := ∅ ;
14      for j ∈ liDo do
            if tk[j] > 0 ∧ q[j] = oq then add j to est fi
        od ;
17      tk[i] := #est + 1.
```

Replacing $ccnt$ by est is only convenient for the proof. It is not yet useful for the algorithm.

As thread i is the only thread that modifies $\mathtt{tk}[i]$, we can represent the lines 22 and 24 of thread i by:

```
22      while ¬ empty(liPri) do
            choose some th ∈ liPri ;
            if tk[th] = 0 ∨ q[th] = oq then
                if tk[th] = 0 ∨ tk[i] · N + i ≤ tk[th] · N + th
                then remove th from liPri fi
24          else
                if wq ≠ oq then remove th from liPri fi
            fi
        od.
```

This transformation is useful to avoid reduncancy and simplify the invariants.

It turns out that, during the transitory period when a thread has executed line 27 and is in loop 30, the old value of wq has a role to play. We therefore introduce a shared ghost variable owq (old waiting queue) which almost always equals wq, but is toggled at the end of loop 30:

```
30          for j ∈ liSw do await ¬ inDo[j] od ;
            owq := wq
```

The proof of mutual exclusion, MX, depends on many invariants, some of them trivial, some of them not so. The invention of these invariants is a creative process that goes partly top-down, partly bottom-up. We give a bottom-up presentation here, so that the reader can convince themselves of the validity of the arguments, although they may fail to see the purpose. We give the invariants symbolic names of the form Xqd with X a capital, d a digit, and a q for the ease of listing and query-replace. We keep gaps in the lists for invariants needed for the half-atomic and nonatomic versions of the algorithm.

The invariants of the first list are more or less obvious observations relating values of variables to locations. In line 13, and only there, thread i sets its shared variable $\mathtt{q}[i] := oq.i$. Moreover, $oq.i$ is modified only in line 12. Therefore, all threads j satisfy

Hq0: j **at** 13 $\vee$ $\mathtt{q}[j] = oq.j$.

In the same way, we have:

Hq1: j **in** $[18 \ldots]$ $\Rightarrow$ $\mathtt{tk}[j] = \#est.j + 1 > 0$,
Hq2: $\mathtt{tk}[j] > 0$ $\Rightarrow$ j **in** $[18 \ldots]$,
Hq3: j **in** $[12 \ldots 18]$ $\Rightarrow$ $\mathtt{inDo}[j]$.

In the postcondition of the loops at 14, 21, and 22, all threads have been treated. Therefore, the corresponding lists are empty:

Iq1: j **in** $[17\ldots] \;\Rightarrow\; liDo.j = \emptyset$,
Iq4: j **in** $[22\ldots] \;\Rightarrow\; liSy.j = \emptyset$,
Iq5: j **in** $[26\ldots] \;\Rightarrow\; liPri.j = \emptyset$.

Finally, thread j never competes with itself:

Iq9: j **in** $[14\ldots] \;\Rightarrow\; j \notin est.j$.

As we want to prove predicate *MX*, we may assume that *MX* is valid in the precondition of every step. This precludes interfering assignments to $\mathtt{wq}$. We therefore have the invariants:

Jq1: j **at** $27 \;\Rightarrow\; oq.j = \mathtt{wq}$,
Jq2: j **in** $[28\ldots] \;\Rightarrow\; oq.j \neq \mathtt{wq}$.

It easily follows from the central code that we have:

Jq3: j **at** $30 \;\Rightarrow\; \mathtt{owq} \neq \mathtt{wq}$,
Jq4: $\mathtt{owq} = \mathtt{wq} \;\vee\; (\exists\, j : j$ **at** $30)$.

We now come to a number of invariants that imply that a certain thread belongs to the waiting queue. The purpose of waiting at 30 becomes explicit in the invariant:

Jq6: j **in** $[13\ldots 18] \;\Rightarrow\; oq.j = \mathtt{wq} \;\vee\; (\exists\, k : k$ **at** $30 \;\wedge\; j \in liSw.k)$.

The equality $oq.j = \mathtt{wq}$ is established by thread j in 12. The only way it can become false, is that some other thread, say k, toggles $\mathtt{wq}$ in 27, but then k arrives at 30, where k cannot leave because of the locking invariant *Hq3*. In other words, the waiting loop at 30 serves to guarantee this invariant. By combining *Jq6* with *Jq3*, we obtain the derived invariant

Dq1: j **in** $[13\ldots 18] \;\Rightarrow\; oq.j = \mathtt{wq} \;\vee\; oq.j = \mathtt{owq}$.

We now come to more difficult invariants about the relations between different threads, and the waiting of thread j in the loops of lines 21 and 22.

Kq0: j **in** $[21\ldots 25] \wedge oq.j = \mathtt{wq} \wedge k$ **in** $[13\ldots 18] \;\Rightarrow\; k \in liSy.j \;\vee\; oq.k = \mathtt{wq}$,
Kq1: j **in** $[22\ldots 27] \wedge oq.j = \mathtt{wq} \wedge k$ **in** $[13\ldots] \;\Rightarrow\; k \in liPri.j \;\vee\; oq.k = \mathtt{wq}$.

Preservation of *Kq0* in 21 follows from *Hq3*. Preservation of *Kq0* when thread i executes 27, follows from *Iq5*, *Jq1*, and *Kq1*, all with $j := i$ (and $k := j$). Preservation of *Kq1* in 27 is proved in the same way. Preservation of *Kq1* in 22 with k **in** $[13\ldots 18]$ follows from *Iq4* and *Kq0*; for k **in** $[18\ldots]$, it follows from *Hq0* and *Hq1*.

It follows from *Kq1*, *Jq1*, and *Iq5*, that the serving queue is empty when some thread is at 27 to swap the queues:

Dq4: j **at** $27 \;\wedge\; k \in [13\ldots] \;\Rightarrow\; oq.k = \mathtt{wq}$.

The invariants *Iq5*, *Jq2*, and *Kq1* imply the derived invariant

Kq1d: j **at** $22 \;\wedge\; k \in [26\ldots] \;\Rightarrow\; k \in liPri.j \;\vee\; oq.j = oq.k$.

This can be proved by showing that the negation of *Kq1d* leads to a contradiction:

$$
\begin{array}{ll}
 & j \text{ at } 22 \;\wedge\; k \in [26\ldots] \;\wedge\; \neg(k \in liPri.j \;\vee\; oq.j = oq.k) \\
\Rightarrow & \{\ Kq1\ \} \\
 & j \text{ at } 22 \;\wedge\; k \in [26\ldots] \;\wedge\; oq.j \neq \mathtt{wq} \;\wedge\; oq.k = \mathtt{wq} \\
\Rightarrow & \{\ Jq2 \text{ with } j := k\ \}
\end{array}
$$

$$
\begin{array}{l}
\quad j \textbf{ at } 22 \ \wedge\ k \in [26 \ldots 27] \ \wedge\ oq.j \neq \mathtt{wq} \ \wedge\ oq.k = \mathtt{wq} \\
\Rightarrow\ \{\, Kq1 \text{ with } j \text{ and } k \text{ swapped } \} \\
\quad k \in [26 \ldots 27] \ \wedge\ j \in liPri.k \\
\Rightarrow\ \{\, Iq5 \text{ with } j := k \,\} \\
\quad false.
\end{array}
$$

Predicate *Kq1d* implies that when thread k is in the critical section, and thread j is about to enter the critical section, the two threads are in the same queue. It remains to compare their token values.

2.6 Correctness: the token values

The token computation in loop 14 is governed by the invariant:

$$Lq0: \qquad j \textbf{ in } [14 \ldots 18] \ \wedge\ k \in est.j \ \Rightarrow\ oq.j = oq.k \ \wedge\ k \textbf{ in } [18 \ldots 30].$$

Preservation of this predicate follows from *Hq0*, *Hq2*, *Jq2*, *Jq6*, and *MX*.

In *Lq0*, when thread j leaves 18 and goes to 21, thread k may proceed to 31, and then exit. This does not happen, however, when j is in the waiting queue or in the old waiting queue. This is expressed in the invariant:

$$
\begin{array}{ll}
Lq2: & j \textbf{ in } [21 \ldots 30] \ \wedge\ (oq.j = \mathtt{wq} \ \vee\ oq.j = \mathtt{owq}) \ \wedge\ k \in est.j \\
& \Rightarrow oq.j = oq.k \ \wedge\ k \textbf{ in } [18 \ldots 30].
\end{array}
$$

At this point, unfortunately, we need a third free variable (h) that ranges over threads, because we want to compare $est.j$ and $est.k$, and the elements of these sets are threads. The invariants *Dq1* and *Lq2* imply:

$$
\begin{array}{l}
j \textbf{ in } [21 \ldots 30] \ \wedge\ h \textbf{ in } [13 \ldots 18] \ \wedge\ oq.j = oq.h \ \wedge\ k \in est.j \\
\Rightarrow oq.j = oq.k \ \wedge\ k \textbf{ in } [18 \ldots 30].
\end{array}
$$

On the other hand, the invariants *Jq4*, *MX*, *Dq1*, and *Jq2* together imply:

$$j \textbf{ in } [31 \ldots] \ \wedge\ h \textbf{ in } [13 \ldots 18] \ \Rightarrow\ oq.h = \mathtt{owq} = \mathtt{wq} \neq oq.j.$$

These two predicates conjoined imply the derived invariant:

$$
\begin{array}{ll}
Lq2d: & j \textbf{ in } [21 \ldots] \ \wedge\ h \textbf{ in } [13 \ldots 18] \ \wedge\ oq.j = oq.h \ \wedge\ k \in est.j \\
& \Rightarrow oq.j = oq.k \ \wedge\ k \textbf{ in } [18 \ldots 30].
\end{array}
$$

This is used to prove the invariant:

$$
\begin{array}{ll}
Lq3: & j \textbf{ in } [21 \ldots] \ \wedge\ k \textbf{ in } [14 \ldots 18] \ \wedge\ oq.j = oq.k \\
& \Rightarrow k \in liSy.j \ \vee\ \{j\} \cup est.j \subseteq liDo.k \cup est.k.
\end{array}
$$

This is a critical invariant needed for token comparison. In view of the token comparison in line 22, we define the state functions *tkk* by

$$tkk.j = (\#est.j + 1) \cdot N + j.$$

Predicate *Hq1* implies that

$$j \textbf{ in } [18 \ldots] \ \Rightarrow\ tkk.j = \mathtt{tk}[j] \cdot N + j.$$

Using *Iq1* and *Iq9*, the invariant *Lq3* implies

$$
\begin{array}{ll}
Lq3d: & j \textbf{ in } [22 \ldots] \ \wedge\ k \textbf{ in } [17 \ldots 18] \ \wedge\ oq.j = oq.k \\
& \Rightarrow tkk.j < tkk.k.
\end{array}
$$

We can extend this beyond line 18 to the invariants:

$$\begin{array}{ll} \mathit{Nq1:} & j \ \mathbf{at}\ 24 \ \wedge\ k = \mathit{th}.j \ \wedge\ k \ \mathbf{in}\ [18 \ldots] \ \wedge\ \mathit{oq}.j = \mathit{oq}.k \\ & \Rightarrow \mathit{tkk}.j < \mathit{tkk}.k\ , \\ \mathit{Nq3:} & j \ \mathbf{in}\ [22 \ldots] \ \wedge\ k \ \mathbf{in}\ [18 \ldots] \ \wedge\ \mathit{oq}.j = \mathit{oq}.k \ \wedge\ k \notin \mathit{liPri}.j \\ & \Rightarrow j = k \ \vee\ \mathit{tkk}.j < \mathit{tkk}.k. \end{array}$$

Now finally, we can prove *MX*, i.e., mutual exclusion, using the invariants *Iq5*, *Kq1d*, and *Nq3* (twice).

2.7 Correctness: FCFS

As the proof obligation *FCFS* is formalized in terms of the ghost variable `predec`, we need some invariants about `predec`. The starting point is:

$$\begin{array}{ll} \mathit{Pq0:} & j \ \mathbf{in}\ [14 \ldots 18] \ \wedge\ k \in \mathtt{predec}[j] \ \wedge\ \mathit{oq}.j = \mathit{oq}.k \\ & \Rightarrow\ \{k\} \cup \mathit{est}.k \subseteq \mathit{est}.j \cup \mathit{liDo}.j. \end{array}$$

This is proved by means of *Lq2d*, *Hq0*, *Hq1*, and the new predicates

$$\begin{array}{ll} \mathit{Pq1:} & k \in \mathtt{predec}[j] \ \Rightarrow\ k \ \mathbf{in}\ [21 \ldots]\ , \\ \mathit{Pq1d:} & j \ \mathbf{at}\ 13 \ \wedge\ k \in \mathtt{predec}[j] \ \wedge\ \mathit{oq}.j = \mathit{oq}.k \ \Rightarrow\ j \notin \mathit{est}.k. \end{array}$$

Predicate *Pq1* is inductive. Predicate *Pq1d* follows from *Pq1* and *Lq2d*. Note that *Pq0* is strikingly similar to *Lq3*, but the antecedent of *Pq0* is stronger because of *Pq1*, and the consequent is stronger because of the absence of the disjunct $k \in \mathit{liSy}.j$.

Using *Pq0*, *Pq1*, *Iq1*, and *Iq9*, we obtain the first token comparison result for `predec`:

$$\mathit{Pq0d:} \quad j \ \mathbf{in}\ [17 \ldots 18] \ \wedge\ k \in \mathtt{predec}[j] \ \wedge\ \mathit{oq}.j = \mathit{oq}.k \ \Rightarrow\ \mathit{tkk}.k < \mathit{tkk}.j.$$

This together with *Pq1* is used to prove the next invariant:

$$\mathit{Pq2:} \quad j \ \mathbf{in}\ [21 \ldots] \ \wedge\ k \in \mathtt{predec}[j] \ \wedge\ \mathit{oq}.j = \mathit{oq}.k \ \Rightarrow\ \mathit{tkk}.k < \mathit{tkk}.j.$$

The goal is now in view with the invariant

$$\mathit{Pq3:} \quad j \ \mathbf{in}\ [22 \ldots] \ \Rightarrow\ \mathtt{predec}[j] \subseteq \mathit{liPri}.j.$$

In order to prove this invariant, however, we also need the invariants:

$$\begin{array}{ll} \mathit{Pq4:} & j \notin \mathtt{predec}[j]\ , \\ \mathit{Pq5:} & j \ \mathbf{in}\ [13 \ldots] \ \wedge\ k \in \mathtt{predec}[j] \ \Rightarrow\ \mathit{oq}.j = \mathtt{wq} \ \vee\ \mathit{oq}.j = \mathit{oq}.k\ , \\ \mathit{Pq7:} & j \ \mathbf{at}\ 24 \ \wedge\ k = \mathit{thr}.j \in \mathtt{predec}[j] \ \Rightarrow\ \mathit{oq}.j \neq \mathit{oq}.k. \end{array}$$

Predicate *Pq4* is clearly inductive. Preservation of *Pq5* at 12 and 27 follows from *Pq1* and *Dq4*. Preservation of *Pq7* at 12 and 22 follows from *Pq1* and *Pq5*.

Finally, our proof obligation *FCFS* follows easily from *Pq3* and *Iq5*.

Strictly speaking, it is only the conjunction of all invariants mentioned that is truly invariant. For instance, in the proof of invariance of *Jq1*, we need that *MX* holds in the precondition. The conjunction of the invariants is called *GlobInv*. Summarizing the above results, we proved with the proof assistant PVS that *GlobInv* is preserved by every step of the algorithm. We also proved that *GlobInv* implies *MX* and *FCFS*, and that it holds initially. By induction, it follows that *GlobInv* remains valid throughout any computation.

3 The half-atomic dual bakery algorithm

Recall that a nonatomic variable may return any value of the appropriate type, when it is read while being written. When aiming at an algorithm for nonatomic variables we need to spread the access to vulnerable information. We therefore split the pairs $\mathtt{tq}[j] = (\mathtt{tk}[j], \mathtt{q}[j])$ which were accessed as single variables in the previous section. This means that the commands 13, 14, 17, 22 and 33 have to be reconsidered. We thus first treat a "half-atomic" version of our bakery algorithm, in which the pairs $(\mathtt{tk}[j], \mathtt{q}[j])$ have been split, but all variables are still atomic.

We split the access to these variables in Sect. 3.1. This introduces a race condition, which is dealt with in Sect. 3.2.

3.1 Splitting access

The pairs $(\mathtt{tk}[j], \mathtt{q}[j])$ are inspected in lines 14 and 22. In both lines, we choose to inspect tk before inspecting q. We reuse the private variable *th* for the chosen thread. Command 14 becomes

```
14      while ¬ empty(liDo) do
            choose some th in liDo ;
            if tk[th] ≠ 0 then
15              if q[th] = oq then add th to est fi ;
            fi ;
            remove th from liDo
        od
```

We remove thread *th* from *liDo* after the conditionals but, if $\mathtt{tk}[th] = 0$, we regard the removal as part of the atomic command 14, because command 15 can be skipped. This is done to get the invariant *Lq3* as elegant as possible.

In the loop of 22, the token comparison is done in 22, the queues are compared in 23, and the queues are compared with wq in 24 (as before):

```
22      while ¬ empty(liPri) do
            choose some th ∈ liPri ;
            if tk[th] = 0 then remove th from liPri
            else
                prio := (tk[i] · N + i < tk[th] · N + th) ;
23              if q[th] = oq then
                    if prio then remove th from liPri fi
                else
24                  if wq ≠ oq then remove th from liPri fi
                fi
            fi
        od
```

In 22, the shared token $\mathtt{tk}[th]$ is read atomically and, if necessary, compared with the own token. In 23, the queue of *th* is read and compared with the own queue *oq*. In 24, the current queue is read. Note that control jumps back to 22 from the **then** branches of both 22 and 23.

3.2 Race condition

When we try to reuse the proofs of the invariants of Sect. 2.5 for the current version of the algorithm, all goes well until the invariant *Lq0*.

Here we arrive at a serious obstruction: it is possible that some thread reads $\mathtt{tk}[k] > 0$ when k is at 31 or 33, and then reads $\mathtt{q}[k]$ when k has proceeded to 13. We have therefore to reckon with the possibility that thread j is at 15 and $\mathtt{tk}[th.j] = 0$. Indeed, the next scenario shows that the present version of the algorithm is incorrect.

Scenario 1 Consider three threads with identifiers $t_0 < t_1 < t_2$.

We start with $\mathtt{wq} = \mathtt{owq} = \mathit{true}$. All threads are at 11. Thread t_2 enters, proceeds to 27, sets $\mathtt{wq} := \mathit{false}$, and reaches 33 with $\mathtt{tk}[t_2] > 0$ and $oq.t_2 = \mathit{true}$. Then thread t_1 enters, gets $\mathtt{q}[t_1] = \mathit{false}$, observes $\mathtt{tk}[t_2] > 0$ at 14, and goes to 15 with $th.t_1 = t_2$. Thread t_2 leaves 33, goes to 11, 12, 13, and sets $\mathtt{q}[t_2] := \mathit{false}$. Thread t_1 reaches 17 with $est.t_1 = \{t_2\}$, sets $\mathtt{tk}[t_1] := 2$, reaches 21. In 21, thread t_1 removes t_0 from $liSy.t_1$ because t_0 is still at 11.

Now thread t_0 enters as well. Threads t_0 and t_2 both reach 17 with $est.t_0 = est.t_2 = \{t_1\}$. Thread t_2 sets $\mathtt{tk}[t_2] := 2$ and $\mathtt{inDo}[t_2] := \mathit{false}$. Thread t_1 in 21 makes $liSy.t_1$ empty by removing t_2 from it, and enters 22. It removes t_0 from $liPri.t_1$ because $\mathtt{tk}[t_0] = 0$. It removes t_2 from $liPri.t_1$ because $2N + t_1 < 2N + t_2$, and goes to 26. Thread t_0 sets $\mathtt{tk}[t_0] := 2$, passes 18, 21, and executes loop 22. It removes t_1 and t_2 from $liPri.t_0$ because all three tokens are equal and $t_0 < t_1 < t_2$, and goes to 26. Finally both t_0 and t_1 proceed from 26 (via 27) to 31, thus violating mutual exclusion.

Mutual exclusion is violated here because thread t_0 can claim priority over thread t_1, but thread t_1 has missed thread t_0 in lines 21 and 22.

We are thus forced to take additional measures. Our first idea was to split the synchronization phase (command 21) of thread i by first waiting for $\neg\mathtt{inDo}[j]$ with $j \in est.i$ and then for $\neg\mathtt{inDo}[j]$ with $j \notin est.i$. It turns out that in this way *MX* can be retained, but *FCFS* fails, i.e., must be weakened.

In some sense, however, the solution is obvious. If we have to deal with the situation that thread j is at 15 and $\mathtt{tk}[th.j] = 0$, we just have to retest $\mathtt{tk}[th.j] \neq 0$. We thus replace the lines 14 and 15 by:

```
14      while ¬ empty(liDo) do
            choose some th in liDo ;
            if tk[th] ≠ 0 then
15              if q[th] = oq then
16                  if tk[th] ≠ 0 then add th to est fi ;
            fi ;
            remove th from liDo
        od
```

In order to show that this indeed solves the problem, we first prove the invariants

Kq4: j **at** 15 $\wedge$ $th.j = k$ $\wedge$ $\mathtt{q}[k] = oq.j$ $\wedge$ $\mathtt{tk}[k] = 0$
$\Rightarrow$ $oq.j = oq.k$ $\wedge$ k **in** $[13 \ldots]$,

Kq6: j **at** 16 $\wedge$ $th.j = k$ $\Rightarrow$ $oq.j = oq.k$ $\wedge$ k **in** $[13 \ldots]$.

Using this, we can prove *Lq0*, *Lq2*, *Lq2d*, *Lq3*, *Nq1*, *Nq3* just as in Sect. 2.6. Yet, for preservation of *Nq3* at 23, we need the additional invariant:

Nq0: j **at** 23 $\wedge$ $prio.j$ $\wedge$ $k = th.j$ $\wedge$ k **in** $[18 \ldots]$ $\wedge$ $oq.j = oq.k$
$\Rightarrow$ $tkk.j < tkk.k$.

The treatment of *FCFS* goes precisely as in Sect. 2.7 but, for the preservation of *Pq3* at command 23, we need the additional invariant:

Pq8: $\quad j$ **at** $23 \ \wedge \ k = th.j \in \mathtt{predec}[j] \ \wedge \ oq.j = oq.k \ \Rightarrow \ \neg prio.j.$

Preservation of *Pq8* at lines 12 and 22 follows from *Pq2*, *Hq1*, and *Pq1*.

Summarizing, the half-atomic algorithm for mutual exclusion consists of the Doorway, Waiting, and Central code of Sect. 2.2, with line 14 split into 14, 15, and 16, and line 22 split into 22 and 23, as shown in Sect. 3.1. Again, we used PVS to verify that the conjunction of the invariants holds initially and is preserved under every step, and that it implies *MX* and *FCFS*.

4 The nonatomic dual bakery algorithm

In this section, we finally present our mutual exclusion algorithm for nonatomic variables. Running ahead of several adaptations needed for the nonatomicity, we present the complete algorithm in Fig. 1. As before, it consists of four program fragments: Doorway (D), Synchronization (S), Waiting (W), and Central code (C).

We present the algorithm and prove mutual exclusion in Sect. 4.1. In Sect. 4.2, we show that the algorithm does not satisfy FCFS, but only some approximation of it. In Sect. 4.3, we prove bounded overtaking: when some thread j has reached line 13, no other thread will reach the critical section more than two times before thread j has done so.

4.1 The algorithm and mutual exclusion

Up to this point, we have the shared Boolean variable `wq`, and the shared arrays `tk`, `q`, `inDo`. We allow these variables to be safe instead of atomic. It turns out that the nonatomicity introduces three problems. The flickering of assignment $\mathtt{tk}[i] := 0$ in line 33 introduces the possibility of interferences, which need to be precluded by means of a new waiting condition. The second problem is the identity of the waiting queue, both at the point where it is read (line 12), and at the point where it is modified (line 27). At line 12, a minor modification of the code saves the day. The nonatomic swapping of the queues at line 27 requires a new Boolean guard, and yet another waiting condition. The third problem is caused by the flickering assignment to $\mathtt{tk}[p]$ in line 17. This can lead to violations of mutual exclusion, see Scenario 2 below. In order to eliminate these possibilities, we make the nondeterministic order of the synchronization loop more deterministic. It seems unavoidable, however, that this flickering can lead to violations of FCFS.

To be concrete, we now assume that the shared variables `tk`, `q`, `wq`, `inDo` are all safe instead of atomic, see Sect. 1.6. This means that the assignments to them in lines 11, 13, 17, 18, 27, and 33 are all flickering. Note that the elements of `tk`, `q`, and `inDo` are output variables, but that `wq` is a safe shared variable that can be written by different threads. We thus have to prove that it is written under mutual exclusion. Fortunately, this is already included in our proof obligation *MX*, because `wq` is only written at line $27 \in [26 \ldots]$.

We first discuss the doorway code D, lines 11–18. In the lines 11, 17, and 18, the assignments to $\mathtt{inDo}[i]$ and $\mathtt{tk}[i]$ have been made flickering. In lines 12 and 13, `wq` is read. Deviating from the code in Sect. 2.2, we do a flickering assignment to $\mathtt{q}[i]$ at 13 only if $\mathtt{q}[i]$ needs to be changed. This is necessary for correctness.

In the language C, one can use the conditional-and operator && and the standard interpretation of integers as Booleans to encode the loop 14–16 by

```
    initally
          ¬ inSw  ∧  wq = owq = nwq
          ∧ (∀ j : j at 10 ∧ ¬inDo[j] ∧ ¬inEx[j] ∧ tk[j] = 0 ∧ q[j] = oq.j ∧ predec[j] = ∅) .

    thread (i : 0 ≤ i < N) =
 10       NCS ; predec[i] := {j | j in [21 ... 32]} ;
D: 11     (flickering) inDo[i] := true ;
 12       liDo := Thread \ {i} ; est := ∅ ;
          if oq ≠ wq then
              oq := wq ;
 13           (flickering) q[i] := oq
          fi ;
 14       while ¬ empty(liDo) do
              choose some th in liDo ;
              if tk[th] ≠ 0 then
 15               if q[th] = oq then
 16                   if tk[th] ≠ 0 then add th to est fi ;
              fi ;
              remove th from liDo
          od ;
 17       (flickering) tk[i] := #est + 1 ;
 18       (flickering) inDo[i] := false ;
S: 19     await ¬ inSw ; liSy := Thread \ {i} ;
 20       for j ∈ liSy with j ∈ est do await ¬ inDo[j] od ;
 21       for j ∈ liSy do await ¬ inDo[j] od ;
          liPri := Thread \ {i} ;
W: 22     while liPri ≠ ∅ do
              choose some th ∈ liPri ;
              if tk[th] ≠ 0 then
                  prio := (tk[i] · N + i < tk[th] · N + th) ;
 23               if q[th] = oq then
                      if ¬ prio then goto 22 fi
                  else
 24                   if wq ≠ oq then remove th from liPri fi ;
                      goto 22
                  fi
              fi ;
 25           await ¬ inEx[th] ; remove th from liPri
          od ;
C: 26     if wq ≠ oq then admit := admit \ {i}
          else
 27           (flickering) inSw := true ; nwq := ¬ nwq ; liSw := Thread \ {i} ;
 28           (flickering) wq := ¬ oq ;
 29           (flickering) inSw := false ;
 30           for j ∈ liSw do await ¬ inDo[j] od ;
              owq := nwq ; admit := Thread \ {i}
          fi ;
 31       critical section ;
 32       (flickering) inEx[i] := true ;
          for all j do predec[j] := predec[j] \ {i} od ;
 33       (flickering) tk[i] := 0 ;
 34       (flickering) inEx[i] := false ; cnt++ ; goto 10 .
    Ghost variables are: predec, admit, nwq, owq, cnt.
```

Fig. 1 The nonatomic dual bakery algorithm

```
for (th = 0 ; th < N ; th++)
if (tk[th] && q[th] == oq && tk[th]) est[th] = 1;
```

The flickering of inDo in line 18 implies that *Hq3* needs to be weakened to

Hq3n: j **in** $[12 \ldots 17] \Rightarrow$ inDo$[j]$.

It follows that we also have to weaken some other invariants: in *Jq6*, *Lq0*, *Lq3* we have to replace the regions $[13 \ldots 18]$ and $[14 \ldots 18]$ by $[13 \ldots 17]$ and $[14 \ldots 17]$, respectively. In *Lq1*, the region of j must be extended to $[18 \ldots 19]$.

The flickering assignment to wq in line 27, where the queues are swapped, is supposed to be done under mutual exclusion. It suffices therefore to introduce a single safe shared Boolean guardian:

var inSw : (safe) $\mathbb{B} := \mathit{false}$

The synchronization phase S: lines 19, 21 now begins with waiting for this guardian to be false.

The flickering of tk[i] in 17 can lead to violations of mutual exclusion, as shown in the following scenario.

Scenario 2 Consider three threads with identifiers $t_0 < t_1 < t_2$. We start with wq = owq = *true*. All threads are at 11. Thread t_2 enters and reaches the assignment to tk[t_2] with $est.t_2$ empty. It starts setting tk[t_2] := 1 flickeringly. Then thread t_1 enters, traverses the doorway, obtains $est.t_1 = \{t_2\}$, sets tk[t_1] := 2, observes ¬ inDo[t_0], and removes t_0 from its list $liSy.t_1$. Then t_2 flickers back with tk[t_2] := 0; thread t_0 enters, reaches the assignment to tk[t_0] with $est.t_0 = \{t_1\}$. Then thread t_2 completes the assignments tk[t_2] := 1 and sets inDo[t_2] to false. Thread t_1 completes its synchronization, and starts its waiting loop with observing that tk[t_0] = 0, and removes t_0 from $liPri.t_1$. Thread t_0 sets tk[t_0] := 2 and completes the doorway. Threads t_2 and t_0 complete synchronization. Thread t_2 enters and completes CS. Then thread t_0 reaches enters *CS* because $t_0 < t_1$. Thread t_1 also enters *CS* because t_0 is not in $liPri.t_1$, thus violating mutual exclusion.

The problem in the above scenario comes from the possibility that, in synchrionization, thread t_1 removes t_0 from $liSy.t_1$ before it removes t_2 from it, while t_2 is in $est.t_1$ and t_0 is not. We therefore decide that synchronizing threads wait first for the elements of *est* to leave the guarded region 12–17, and subsequently for the other threads. We thus reduce the nondeterminism by splitting the synchronization as follows:

Synchronization code for thread i:

19 **await** ¬ inSw ; $liSy := \mathit{Thread} \setminus \{i\}$;
20 **for** $j \in liSy$ **with** $j \in est$ **do await** ¬ inDo[j] **od** ;
21 **for** $j \in liSy$ **do await** ¬ inDo[j] **od**

Note that loop 20 removes all elements of *est* from *liSy*. This gives the obvious new invariants:

Iq2: j **at** 20 $\Rightarrow$ $liSy.j \cup est.j = \mathit{Thread} \setminus \{i\}$,
Iq3: j **in** [21 . . .] $\Rightarrow$ $liSy.j \cap est.j = \emptyset$.

We need *Iq2* to prove the invariance of *Kq0* at 21.

Making the assignment to tk[i] in line 33 flickering has also serious consequences. It allows a thread k to remain hidden at 33 with $tk[k] = 0$ and arbitrary q[k], and then suddenly to emerge with $tk[k] > 0$ again. This would enable scenarios with some thread j at 14 and $k \in est.j$ with $oq.j \neq oq.k$. We thus have to preclude threads from entering the critical section when some thread is still flickering at 33. To guard the assignment at 33, we introduce another array of safe shared bits (initially false):

var inEx : **array** [N] **of** (safe) $\mathbb{B}$.

Waiting for this guardian is inserted at line 25. Note, however, that thread i only waits at 25 when it has observed tk[th] = 0 or $prio \wedge$ q[th] = oq. Also note that, in the programming language C, the commands "**goto** 22" in lines 23 and 24 can be coded as **continue**.

In the central code C: lines 26–34, we introduce a shared ghost variable nwq to replace wq in all invariants except one. The value of wq is determined by nwq via the invariant:

Jq5: wq = nwq $\vee$ ($\exists$ j : j **at** 28).

Indeed, as shown by the code, wq flickers when some thread j is at 28.

The guards inEx[i] were introduced to include line 33 in the critical section. We cannot expect line 34 to belong to the critical section because of the flickering of inEx[i]. We therefore redefine predicate *MX* (mutual exclusion) by

MX: $\quad j$ **in** $[26 \ldots 33] \ \wedge \ k$ **in** $[26 \ldots 33] \ \Rightarrow \ j = k.$

The sole purpose of the guard inSw tested in 19 is to ensure the invariant

Kq3: $\quad j$ **in** $[21 \ldots 33] \ \wedge \ k$ **at** $28 \ \Rightarrow \ oq.j \neq \mathtt{nwq}.$

This invariant is needed because, when some thread k is at 28 so that wq can be flickering, another thread j might see the new value of wq in 12. In absence of the test at 19, thread j might enter the waiting loop and pass 24 because thread k resets wq to the old value. Then threads j and k could enter the critical section together and violate mutual exclusion.

For the algorithm of Fig. 1, we verified with PVS that the conjunction of the invariants holds initially and is preserved in every step. It follows that it satisfies mutual exclusion. The waiting loops at 21 and 30 can be implemented as sketched in Sect. 3.

To summarize the amount of waiting in the present algorithm, the waiting loops of 19, 21, and 22–25 also occur in the bakery algorithms of Lamport and Taubenfeld. The waiting at 19 and 25 is needed to accommodate the flickering assignments at 28 and 33, respectively. The waiting loop at 30 is needed because of our token definition (1).

The complete list of supporting invariants for the nonatomic dual bakery algorithm is given in the appendix. The importance of the guards inEx[i] appears in the invariants that apply to regions ending in 32: *Hq1*, *Kq2*, *Nq0*, *Nq2*, as opposed to those ending in 33: *MX*, *Hq2*, *Jq2*, *Kq1*, *Kq3*, *Nq3*.

4.2 Quasi-FCFS

As announced, the nonatomic algorithm does not satisfy FCFS. This is shown as follows. According to the definition of FCFS, the doorway must be wait-free. This implies that we can take the doorway not larger than the program fragment 10–18. Now there are two scenarios that violate FCFS. They both allow a thread t_0 to reach line 19 or even 20, after which a second thread t_1 enters the doorway, while thread t_1 reaches *CS* before t_0. In either case, the problem comes from a third thread t_2 that enters before t_0, and is doing a flickering assignment at line 28 or line 17. Both scenarios begin with all threads idle at 10, and $\mathtt{wq} = \mathit{false}$.

Scenario 3 Thread t_2 enters and proceeds to line 28, where it starts setting $\mathtt{wq} := \mathit{true}$, flickeringly. Thread t_0 enters and reaches line 19 with $oq.t_0 = \mathit{true}$. Then thread t_1 enters and reaches line 19 with $oq.t_1 = \mathit{false}$. Then thread t_2 completes its assignment $\mathtt{wq} := \mathit{true}$ at 28, goes to *CS* and exits. Threads t_0 and t_1 pass 19 and reach loop 22. Thread t_1 enters *CS* before t_0 because it gets priority in line 24. □

Scenario 4 Thread t_2 enters and proceeds to line 17, where it starts setting $\mathtt{tk}[t_2] := 1$, flickeringly. Thread t_0 reaches line 20 with $est.t_0 = \{t_2\}$ and $\mathtt{tk}[t_0] = 2$. Then thread t_1 enters and reaches line 20 with $est.t_1 = \{t_0\}$ and $\mathtt{tk}[t_1] = 2$. Then thread t_2 completes its assignment at 17 and proceeds to line 20. All three threads proceed to loop 22, with $oq = \mathit{false}$. Thread t_2 enters *CS* first. We now assume that the thread identifier of t_1 is lower than the identifier of t_0. Therefore thread t_1 enters *CS* before t_0 because it gets priority in line 23. □

After these sobering scenarios, let us turn to the positive side. We have three results on safety of the algorithm that approximate FCFS.

Firstly, when the shared variables behave atomically, the system satisfies FCFS, because the half-atomic version has this property and, when the assignments are interpreted atomically, the nonatomic version is an implementation (with additional waiting) of the half-atomic one.

The second result is that when a thread i enters at 11 when thread j has passed the region 11–20, then thread j will enter *CS* before thread i. This proof goes in the same way as the FCFS proof of Sect. 2.7. It uses the ghost variable `predec`. When thread i enters, it sets `predec`$[i]$ to the set of threads j with j **in** $[21 \ldots 33]$. Whenever some thread i leaves 33, it removes itself from all sets `predec`$[j]$. We prove the invariant

qFCFS: $\quad j$ **in** $[26 \ldots] \;\Rightarrow\; \texttt{predec}[j] = \emptyset$.

The proof relies on the invariants *Pq0* up to *Pq8* as given in the Appendix. This does not prove FCFS because the region 11–20 is not wait-free. A referee suggested to describe this property as "FCFS with an unbounded doorway".

4.3 Bounded overtaking

For threads that have entered the doorway up to line 13 but have not yet passed the synchronization 19–20, we can approximate FCFS by the weaker guarantee of bounded overtaking.

We prove that, while thread j is in the region 13–30, the number of times thread k enters the critical section is bounded by 2.

This result is obtained by means of a variant function. We first introduce a new ghost variable `admit` as an upper bound for the serving queue. Initially, `admit` is arbitrary (e.g. empty). The ghost variable `admit` is modified only when the acting thread i enters the critical section from the lines 26 and 30, as shown in Fig. 1.

The fact that `admit` is an upper bound of the serving queue is expressed by the invariant

Qq0: $\quad j$ **in** $[13 \ldots 30] \;\Rightarrow\; oq.j = \texttt{nwq} \;\vee\; oq.j = \texttt{owq} \;\vee\; j \in \texttt{admit}$.

This invariant is not very difficult to prove.

In order to prove the upper bound 2 for the number of times thread k enters *CS* while thread j is in region 13–30, we define the variant function for bounded overtaking

$$bovf(j,k) = (oq.j = \texttt{nwq} \wedge \texttt{nwq} \neq \texttt{owq}\;?\;2 :\; |oq.j = \texttt{owq}| + |k \in \texttt{admit}|)\,,$$

where $|b| = (b\;?\;1 : 0)$ for any Boolean value b. It is clear that $bovf(j,k)$ can only be 2, 1, or 0. The design of this complicated function is based on a careful analysis of how thread k can repeatedly enter *CS* while thread j remains in 13–30, in relation to the values of `wq`, `nwq`, `owq`, $oq.j$ and $oq.k$. Its aim is the following result:

Theorem 1 *While thread j is in region* 13–30*, the value $bovf(j,k)$ never increases. It is modified only when some thread executes line* 26 *or* 30*. Whenever thread k enters the critical section (from line* 26 *or* 30*), $bovf(j,k)$ decreases.*

This result is proved using the invariants *Qq0* and *Jq3*, and the relations between the Booleans `owq`, `nwq`, and `wq`.

As $bovf(j,k)$ is bounded by $0 \leq bovf(j,k) \leq 2$, Theorem 1 proves that, while thread j is in region 13–30, thread k enters *CS* at most twice.

5 Progress

In this section, we formally prove that the nonatomic algorithm of Sect. 4 satisfies liveness under weak fairness. This is prepared in Sect. 5.1 by discussing forward steps and weak

disabledness. In Sect. 5.2, we construct a variant function that never decreases, and that increases under most of the forward steps of the thread involved. In Sect. 5.3, we analyse the thread selection in loop 22–25.

Finally, in Sect. 5.4, we introduce our version of temporal logic, define weak fairness, and prove that in every weakly-fair execution, every thread is eventually always at *NCS*.

5.1 Weak disabledness

For the sake of progress and liveness, we partition the steps in two types: forward steps and other steps. Progress and liveness of thread i depends on its forward steps. The other steps are merely allowed, and may be regarded as done by an uncontrolable environment.

The *forward* steps of thread i are those that begin in states with $pc.i \neq 10$, and that modify $pc.i$ or remove an element from one of the sets $liDo.i$, $liSy.i$, $liPri.i$, or $liSw.i$. The *other* steps are the step from line 10 that makes thread i compete, and all flickering steps of i. One could argue that the critical section *CS* is also executed by the environment of the algorithm, but for the proof of progress we need the assumption that *CS* terminates. We therefore treat *CS* as a forward step.

As announced in Sect. 2.3, we need to discuss the conditions under which forward steps are disabled, in particular for the waiting loops at the lines 20, 21, and 30. For example, strictly speaking, line 20 of Fig. 1 is enabled when there is a thread $j \in liSw \cup est$ with $\neg\ \texttt{inDo}[j]$. We prefer, however, to allow a nondeterministic choice of j followed by waiting for $\neg\ \texttt{inDo}[j]$. This would be enoded better by means of two atomic commands:

```
20    if liSy ∩ est = ∅ then goto 21 else
      choose j ∈ liSy ∩ est ;
      remove j from liSy ; goto 20.1 fi ;
20.1  await ¬ inDo[j] ; goto 20 ;
```

We avoided this complication in Fig. 1 for the sake of clarity, and also because it only matters for liveness.

Our aim is to prove liveness of the algorithm even with this implementation of loop 20, and similar implementations of loops 21 and 30.

Partly also for simplicity, we thus prove progress under the assumption that forward steps of thread i can be disabled when thread i is waiting for some element of the arrays `inDo` or `inEx`, while some (possibly other) element of the array holds *true*. We thus define the condition of *weak disabledness Wdis* by

$$\begin{array}{l} Wdis(i) \ \equiv \\ pc.i = 10 \ \vee \ (pc.i = 19 \ \wedge \ \texttt{inSw}) \\ \ \vee \ (pc.i \in \{20, 21, 30\} \ \wedge \ \exists\ j : \texttt{inDo}[j]) \\ \ \vee \ (pc.i = 25 \ \wedge \ \exists\ j : \texttt{inEx}[j]). \end{array}$$

For the consistency of the PVS code, it is important that we verified that thread i can indeed do a forward step in every state where $Wdis(i)$ is false. The verification of this requires the obvious invariant that $pc.i \in \{10 \ldots 34\}$.

Weak disabledness is sufficient to prove absence of "weak" deadlock: if all threads are weakly disabled, all threads are in the noncritical section *NCS*:

$$(\forall\ i : Wdis(i)) \ \Rightarrow \ j \ \textbf{at}\ 10. \tag{4}$$

This follows from the easy invariants:

Hq5: $\quad \texttt{inEx}[j] \ \Rightarrow \ j \ \textbf{in}\ [32 \ldots]\ ,$

Hq6: $\texttt{inDo}[j] \;\Rightarrow\; j$ **in** $[11 \ldots 18]$,
Hq7: $\texttt{inSw} \;\Rightarrow\; \exists\, j : j$ **in** $[27 \ldots 29]$.

We do not prove formula (4) mechanically because it does not preclude livelock, and it is subsumed by the proof of liveness that is given below.

5.2 A variant function for progress

We construct integer valued state functions that express progress for each thread, and that grow proportionally to the number of times the thread executes the critical section.

This is done in the following way. As shown in Fig. 1, each thread i has a private ghost variable $cnt.i$ which is incremented with 1 whenever thread i executes line 34 and goes back to *NCS*. We now introduce a constant A and the variant function

$$\begin{aligned} avf.i = A \cdot cnt.i &+ (i \textbf{ in } [22 \ldots 25] \,?\, 12 : pc.i - 10) \\ &+ (i \textbf{ in } [13 \ldots] \,?\, 3 \cdot (N - \#liDo.i) : 0) \\ &+ (i \textbf{ in } [20 \ldots] \,?\, N - \#liSy.i : 0) \\ &+ (i \textbf{ in } [28 \ldots] \,?\, N - \#liSw.i : 0). \end{aligned}$$

It is clear that $avf.i$ changes only when thread i itself does a step. We choose $A > 5 \cdot N + 24$. This implies that the jump from line 34 to line 10 increases $avf.i$ and that $avf.i$ never decreases. Indeed, most forward steps of thread i increase $avf.i$. The only exceptions are the steps within the loop 22–25, which keep $avf.i$ constant.

The growth of $avf.i$ is proportional to the growth of $cnt.j$ because of

$$A \cdot cnt.i \le avf.i < A \cdot (cnt.i + 1).$$

5.3 The selection of the thread to enter *CS*

In the loop of lines 22–25, the next thread for *CS* is selected from the serving queue (i.e. with $oq \neq \texttt{nwq}$) with the lowest tkk, or if the serving queue is empty, from the waiting queue with the lowest tkk. We unify this by defining:

$$tkkN.j = tkk.j + (oq.j = \texttt{nwq} \,?\, N^2 + 2N : 0).$$

Note that $tkk.j < N^2 + 2N$ always holds. The second summand therefore implies that the threads in the serving queue have lower values for $tkkN$ than those in the waiting queue.

We define a thread i to be the *elected* if it satisfies the condition

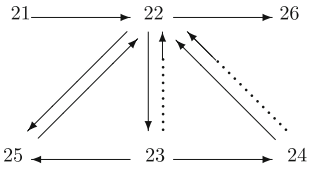
$$i \textbf{ in } [22 \ldots 26] \;\wedge\; (\forall\, j : j \textbf{ in } [22 \ldots 25] \;\Rightarrow\; tkkN.i \le tkkN.j).$$

If there are threads in $[22 \ldots 25]$, there is an elected thread. If it exists, the elected thread is unique because $tkkN.j$ equals j modulo N by construction.

We analyse progress of the elected thread under the simplifying assumption that the state is *quiet*: all threads are at line 10 or in the loop 22–25. More precisely, we prove

Theorem 2 *If at some point in some execution, thread i is elected and the state is quiet and remains quiet, then thread i remains elected and from some point onward thread i does no steps anymore.*

Proof Because the state is and remains quiet, nwq remains constant and for all threads j the values of $oq.j$ and $est.j$, and hence also of $tkk.j$ and $tkkN.j$ remain constant. Therefore, thread i remains elected. To analyse the steps of thread i, we draw the transition diagram of loop 22–25:



Here, the dotted transitions are the backward steps to 22 that do not remove *th.i* from *liPri.i*. This happens from the conditions:

$$\begin{array}{lcl} Nrq24.i & \equiv & pc.i = 24 \ \wedge \ \mathrm{wq} = oq.i \ , \\ Nrq23.i & \equiv & pc.i = 23 \ \wedge \ (\mathrm{q}[th.i] = oq.i \ ? \ \neg prio.i : \mathrm{wq} = oq.i). \end{array}$$

We therefore define the variant function

$$loopvf.i = 25 - pc.i + 4 \cdot \#lipri.i + 3 \cdot |Nrq23.i \vee Nrq24.i|.$$

When the state is quiet, every step of any thread $j \neq i$ keeps $loopvf.i$ constant. When thread i makes a step from 23, 24, or 25, it decrements $loopvf.i$ because $th.i \in liPri.i$ by the invariant *Iq12*. When the state is quiet, and thread i is elected and it does a step from line 22, it also decrements $loopvf.i$. The proof of this requires the invariants *Hq0*, *Hq1*, and *Jq5*. Now, the assertion follows from the obvious lower bound $loopvf.i \geq 0$. □

5.4 Liveness under weak fairness

We finally prove that under the assumption of weak fairness every thread that enters at line 11 eventually comes back to the noncritical section. We need weak fairness for this, because otherwise some thread can monopolize the execution by continuously cycle through loop 22–25 without changing its set *liPri*.

The assumption of *weak fairness* means that we assume that all executions of the system are weakly fair. Recall that an execution is an infinite sequence of states that starts in an initial state and has a step between every pair of subsequent states. An execution is called *weakly fair* iff, whenever from some state onward some thread j can always do a forward step, thread j will eventually do the step. Recall that a *forward* step is a nonflickering step from a location $\neq 10$.

We formalize the setting in a set-theoretic version of temporal logic. Let X be the state space. We identify the set X^ω of the infinite sequences of states with the set of functions $\mathbb{N} \to X$. For a subset $U \subseteq X$, we define $[\![U]\!] \subseteq X^\omega$ as the set of sequences xs with $xs(0) \in U$. For a relation $A \subseteq X^2$, we define $[\![A]\!]_2 \subseteq X^\omega$ as the set of sequences xs with $(xs(0), xs(1)) \in A$.

For $xs \in X^\omega$ and $m \in \mathbb{N}$, we define the shifted sequence $D(m, xs)$ by $D(m, xs)(n) = xs(m + n)$. For a subset $P \subseteq X^\omega$ we define $\Box P$ and $\Diamond P$ as the subsets of X^ω given by

$$\begin{array}{lcl} xs \in \Box P & \equiv & (\forall \, m \in \mathbb{N} : D(m, xs) \in P) \ , \\ xs \in \Diamond P & \equiv & (\exists \, m \in \mathbb{N} : D(m, xs) \in P). \end{array}$$

We read $\Box P$ as *always* P, and $\Diamond P$ as *eventually* P. Sometimes we need to be more explicit than the operator $\Diamond$ allows. We have $\Diamond P = \bigcup_m \Diamond_m P$, where $\Diamond_m$ is given by

$$xs \in \Diamond_m P \ \equiv \ D(m, xs) \in P.$$

We now apply this to the algorithm. We write $init \subseteq X$ for the set of initial states and $step \subseteq X^2$ for the step relation on X. Following [1], we use the convention that relation *step*

is reflexive (contains the identity relation). An *execution* is an infinite sequence of states that starts in an initial state and in which each subsequent pair of states is connected by a step. The set of executions of the algorithm is therefore

$$Ex = [\![\, init \,]\!] \cap \Box [\![\, step \,]\!]_2.$$

By induction, every state in an execution satisfies all invariants proved for the algorithm. For any thread j, the numbers $cnt.j$ and $avf.j$ form nondecreasing sequences along any execution.

We define (j **at** 10) to be the subset of X of the states in which thread j is at line 10. An execution in which thread j is always eventually at *NCS*, is therefore an element of $\Box\Diamond[\![\, j \textbf{ at } 10 \,]\!]$.

Weak fairness is formalized as follows. Let $fw(j) \subseteq X^2$ be the set of forward steps of thread j. Using weak disabledness as defined in Sect. 5.1, we regard an execution as *weakly fair* [24] for thread j iff it belongs to

$$WF(j) = \Box\Diamond[\![\, Wdis(j) \,]\!] \cup \Box\Diamond[\![fw(j) \,]\!]_2.$$

The set of weakly fair executions is defined as the intersection (conjunction):

$$WF = \bigcap\nolimits_j WF(j).$$

At this point, we can state the main liveness result:

Theorem 3 *In every weakly fair execution, every thread j is always eventually at NCS.*

This is expressed in the temporal formula:

$$Ex \cap WF \subseteq \Box\Diamond[\![\, j \textbf{ at } 10 \,]\!]. \tag{5}$$

Proof We define $Bcnt(j)$ to mean that $cnt.j$ is bounded, and $Cavf(j)$ to mean that $avf.j$ is constant:

$$Bcnt(j) = \cup_z \Box[\![\, cnt.j \leq z \,]\!],$$
$$Cavf(j) = \cup_z \Box[\![\, avf.j = z \,]\!].$$

The proof begins with the observation that, if some thread j remains away from line 10, its value $cnt.j$ is bounded, as expressed in

$$Ex \subseteq \Box\Diamond[\![\, j \textbf{ at } 10 \,]\!] \cup Bcnt(j), \tag{6}$$

Because $avf.j$ is nondecreasing and grows proportionally to $cnt.j$, any execution in which $cnt.j$ is bounded, is such that $avf.j$ eventually becomes constant:

$$Ex \cap Bcnt(j) \subseteq \Diamond Cavf(j). \tag{7}$$

When $avf.j$ is constant, $pc.j$ is constant unless j **in** $[22 \ldots 25]$:

$$Ex \cap \Diamond_m Cavf(j) \subseteq \Diamond_m(\Box[\![\, j \textbf{ in } [22 \ldots 25] \,]\!] \cup \textstyle\bigcup_\ell \Box[\![\, j \textbf{ at } \ell \,]\!]).$$

At this point, we apply weak fairness. Because $Wdis(j)$ implies that thread j is at 10 or 30 or in $[19 \ldots 25]$, and because $avf.j$ increases under all forward steps except those within the loop 22–25, we obtain

$$\begin{aligned} &Ex \cap \Diamond_m Cavf(j) \cap WF(j) \\ &\subseteq \Diamond_m(\Box[\![\, j \textbf{ at } 10 \,]\!] \cup \Box[\![\, j \textbf{ in } [19 \ldots 25] \,]\!] \cup \Box[\![\, j \textbf{ at } 30 \,]\!]). \end{aligned} \tag{8}$$

Theorem 1 about bounded overtaking implies that when some thread j is eventually always in $[19 \ldots 30]$, $cnt.k$ is bounded for every thread k:

$$Ex \cap \Diamond\Box[\![\ j \textbf{ in } [19 \ldots 30]\ ]\!] \subseteq Bcnt(k).$$

Using formula (7) and finiteness of the number of threads, we get

$$Ex \cap \Diamond\Box[\![\ j \textbf{ in } [19 \ldots 30]\ ]\!] \subseteq \Diamond \bigcap_k Cavf(k). \tag{9}$$

Formula (8) immediately gives:

$$\begin{aligned} &Ex \cap \Diamond_m \textstyle\bigcap_k Cavf(k) \cap WF \\ &\subseteq\ \Diamond_m \textstyle\bigcap_k (\Box[\![\ k \textbf{ at } 10\ ]\!] \cup \Box[\![\ k \textbf{ in } [19 \ldots 25]\ ]\!] \cup \Box[\![\ k \textbf{ at } 30\ ]\!]). \end{aligned}$$

When all threads are at 10 or 30 or in $[19 \ldots 25]$, it follows from the invariants *Hq5*, *Hq6*, *Hq7*, that all weakly disabled threads are at 10 (see Sect. 5.1). Using weak fairness, we can therefore strengthen the last formula to

$$Ex \cap \Diamond_m \bigcap_k Cavf(k) \cap WF \ \subseteq\ \Diamond_m \Box Quiet, \tag{10}$$

where $Quiet = \bigcap_k [\![\ (k \textbf{ at } 10) \cup (k \textbf{ in } [22 \ldots 25])\ ]\!]$ as used in Sect. 5.3. In quiet states, no threads are weakly disabled. If there is an elected thread, weak fairness therefore guarantees progress for the elected thread. It follows that Theorem 2 implies

$$Ex \cap WF \cap \Diamond_m \Box Quiet \subseteq \Diamond_m \Box[\![\ j \textbf{ at } 10\ ]\!]. \tag{11}$$

Combining the formulas (6), (7), (8), (9), (10), and (11), we finally obtain

$$Ex \cap WF \subseteq \Box\Diamond[\![\ j \textbf{ at } 10\ ]\!] \cup \Diamond\Box[\![\ j \textbf{ at } 10\ ]\!] \subseteq \Box\Diamond[\![\ j \textbf{ at } 10\ ]\!].$$

This proves formula (5) and hence the theorem. □

6 History and PVS proofs

The ideas for the algorithm were developed independently of any proof assistant. When we started to try and prove the safety of the algorithm, we encountered problems in the algorithm, which subsequently led to refutations with only three threads by means of the model checker Spin [13]. When we finally had a version that could not be refuted by Spin, we again started our proof effort with the proof assistant PVS [27].

This led to a proof with PVS of a version of the nonatomic algorithm that required more waiting than the version presented here. We then invented the atomic and the half-atomic versions of the algorithm, primarily to present the algorithm and its proof in a structured way. In this way we arrived at a paper in which we concentrated on safety of the algorithm. This version suggested that the algorithm satisfied FCFS and it lacked a formal proof of liveness. Thanks to critical reviewers, we had the opportunity to reconsider FCFS and to provide a formal proof of liveness. We were only able to provide the latter proof after we had done the same for Lamport's bakery algorithm itself.

The project crucially depends on the proof assistant PVS. A proof assistant like PVS allows the user to redo proofs in different contexts. This enabled us to optimize the algorithm, in its three versions, while keeping the proofs valid. In other words, we developed and applied the design approach sketched in Sect. 1.3 to present and optimize the algorithm, after having proved that the vulnerable information was spread enough.

The translation of pseudocode into PVS is manual, and subject to errors. The translation of a mental concept in machine code and the interpretation of the result always ultimately

depends on human intelligence (J Moore). The user of PVS has the responsibility to check that the responses of the tool correspond to what can be expected from the mental image they have of what has been formalized in the tool. When many different things are proved about the same formal model, it is likely that almost all discrepancies are caught.

Deliberately masking bugs in the translation is useless, because a PVS-expert may find it, an expert in the proof may see an invalid argument, and an expert in the algorithm may find a violating scenario, even without looking at the proof. The PVS dumpfile on our website [12] can be inspected to see the formalization and to verify whether PVS can prove it. This of course requires some fluency with PVS.

The main goal of the tool is to enable the authors to create an exhaustive proof, and to gain confidence about the algorithm in this way. It is up to the reader to judge whether the result is convincing. For this purpose, the reader reads the paper, but he can also try and read the PVS dumpfile, or consult a PVS expert.

In the end, the atomic version has 18 transitions and 28 invariants. The proof takes 111 lemmas, which have been proved interactively. Replaying the proofs on an ordinary laptop takes less than 5 min. The half-atomic version has 21 transitions and 33 invariants. The proof takes 138 lemmas, which can be proved in less than 10 min. The proofs for the atomic version and the half-atomic version only cover the safety properties of mutual exclusion and FCFS.

The nonatomic version has 35 transitions. Here we need 38 invariants for mutual exclusion, 9 invariants for almost-FCFS, 4 invariants for bounded overtaking, and 4 for progress. The proof consists of 296 lemmas that take around 20 min to replay. The PVS proof scripts are available at [12].

Important techniques in the proofs were the introduction of the ghost variables *liSy*, etc., for the sets of threads that had to be treated yet in the loops, and of the ghost variables `nwq` and `owq` to represent the flickering value of the shared variable `wq`. The liveness proof critically depends on three variant functions of a rather different nature.

7 Conclusions

The reader may have noticed that Sect. 4 on the nonatomic version is shorter than the description of the atomic versions in Sects. 2 and 3. This may suggest that when the vulnerable information is spread sufficiently, it is easy to make the variables nonatomic. This is not true. The verifications for the nonatomic case are more complex but, when it turns out that everything is still valid, they are not illuminating to discuss. Indeed, the above statistics of the PVS proof indicate that the nonatomic case is, roughly speaking, twice as difficult as the half-atomic case. For the human reader, however, it may look like more of the same.

Algorithms for concurrency and especially for nonatomic variables are difficult to design. There is only a small number of mutual exclusion algorithms with nonatomic shared variables known [3,5,6,17–21,25,29]. These papers are not all equally explicit about atomicity assumptions and verification. All shared variables in our present algorithm are allowed to be safe, i.e., nonatomic. We have formally and mechanically proved mutual exclusion, almost-FCFS, bounded overtaking, and liveness under weak fairness.

Nonatomic algorithms may become practically relevant now, because several recent systems such as smart-phones, multi-mode handsets, network processors, graphics chips, and other high performance electronic devices use multiport memories, and such memories allow nonatomic accesses through multiple ports [14,31,39].

Appendix: the invariants of the nonatomic version

MX: j **in** $[26 \ldots 33] \ \wedge \ k$ **in** $[26 \ldots 33] \ \Rightarrow \ j = k$,
Hq0: j **at** $13 \ \vee \ \mathtt{q}[j] = oq.j$,
Hq1: j **in** $[18 \ldots 32] \ \Rightarrow \ \mathtt{tk}[j] = \#est.j + 1 > 0$,
Hq2: $\mathtt{tk}[j] > 0 \ \Rightarrow \ j$ **in** $[17 \ldots 33]$,
Hq3: j **in** $[12 \ldots 17] \ \Rightarrow \ \mathtt{inDo}[j]$,
Hq4: j **at** $33 \ \Rightarrow \ \mathtt{inEx}[j]$,
Hq5: $\mathtt{inEx}[j] \ \Rightarrow \ j$ **in** $[32 \ldots]$,
Hq6: $\mathtt{inDo}[j] \ \Rightarrow \ j$ **in** $[11 \ldots 18]$,
Hq7: $\mathtt{inSw} \ \Rightarrow \ \exists \ j : j$ **in** $[27 \ldots 29]$.

Iq0: j **at** $13 \ \Rightarrow \ liDo.j = Thread \setminus \{j\}$,
Iq1: j **in** $[17 \ldots] \ \Rightarrow \ liDo.j = \emptyset$,
Iq2: j **at** $20 \ \Rightarrow \ \{j\} \cup liSy.j \cup est.j = Thread$;
Iq3: j **in** $[21 \ldots] \ \Rightarrow \ liSy.j \cap est.j = \emptyset$,
Iq4: j **in** $[22 \ldots] \ \Rightarrow \ liSy.j = \emptyset$,
Iq5: j **in** $[26 \ldots] \ \Rightarrow \ liPri.j = \emptyset$,
Iq6: j **in** $[28 \ldots 29] \ \Rightarrow \ liSw.j = Thread \setminus \{j\}$,
Iq7: j **at** $13 \ \Rightarrow \ est.j = \emptyset$,
Iq8: j **in** $[15 \ldots 16] \ \Rightarrow \ th.j \neq j$,
Iq9: j **in** $[14 \ldots] \ \Rightarrow \ j \notin est.j$,
Iq10: j **in** $[22 \ldots] \ \Rightarrow \ j \notin liPri.j$,
Iq11: j **in** $[15 \ldots 16] \ \Rightarrow \ th.j \in liDo.j$,
Iq12: j **in** $[23 \ldots 25] \ \Rightarrow \ th.j \in liPri.j$,
Iq13: j **in** $[10 \ldots 34]$.

Jq0: j **at** $28 \ \Rightarrow \ \mathtt{inSw}$,
Jq1: j **in** $27 \ \Rightarrow \ oq.j = \mathtt{nwq}$,
Jq2: j **in** $[28 \ldots 33] \ \Rightarrow \ oq.j \neq \mathtt{nwq}$,
Jq3: j **in** $[28 \ldots 30] \ \Rightarrow \ \mathtt{owq} \neq \mathtt{nwq}$,
Jq4: $\mathtt{owq} = \mathtt{nwq} \ \vee \ (\exists \ j : j$ **in** $[28 \ldots 30])$,
Jq5: $\mathtt{wq} = \mathtt{nwq} \ \vee \ (\exists \ j : j$ **at** $28)$,
Jq6: j **in** $[13 \ldots 17] \ \Rightarrow \ oq.j = \mathtt{nwq} \ \vee \ (\exists \ k : k$ **in** $[28 \ldots 30] \ \wedge \ j \in liSw.k)$.

Kq0: j **in** $[21 \ldots 25] \ \wedge \ oq.j = \mathtt{nwq} \ \wedge \ k$ **in** $[13 \ldots 17]$
$\Rightarrow k \in liSy.j \ \vee \ oq.k = \mathtt{nwq}$,
Kq1: j **in** $[22 \ldots 27] \ \wedge \ oq.j = \mathtt{nwq} \ \wedge \ k$ **in** $[13 \ldots 33]$
$\Rightarrow k \in liPri.j \ \vee \ oq.k = \mathtt{nwq}$,
Kq2: j **at** $25 \ \wedge \ oq.j = \mathtt{nwq} \ \wedge \ k = th.j \ \wedge \ k$ **in** $[13 \ldots 32] \ \Rightarrow \ oq.k = \mathtt{nwq}$,
Kq3: j **in** $[20 \ldots 33] \ \wedge \ k$ **at** $28 \ \Rightarrow \ oq.j \neq \mathtt{nwq}$,
Kq4: j **at** $15 \ \wedge \ th.j = k \ \wedge \ \mathtt{q}[k] = oq.j \ \wedge \ \mathtt{tk}[k] = 0 \ \Rightarrow \ k$ **in** $[13 \ldots 17]$,
Kq5: j **at** $15 \ \wedge \ th.j = k \ \wedge \ k$ **at** $13 \ \Rightarrow \ oq.j = oq.k$,
Kq6: j **at** $16 \ \wedge \ th.j = k \ \Rightarrow \ oq.j = oq.k \ \wedge \ k$ **in** $[13 \ldots 33]$.

Lq0: j **in** $[14 \ldots 17] \ \wedge \ k \in est.j \ \Rightarrow \ oq.j = oq.k \ \wedge \ k$ **in** $[17 \ldots 30]$,
Lq1: j **in** $[20 \ldots 30] \ \wedge \ (oq.j = \mathtt{nwq} \ \vee \ oq.j = \mathtt{owq}) \ \wedge \ k \in est.j \ \wedge \ k$ **at** 17
$\Rightarrow k \in liSy.j$,
Lq2: j **in** $[18 \ldots 30] \ \wedge \ (oq.j = \mathtt{nwq} \ \vee \ oq.j = \mathtt{owq}) \ \wedge \ k \in est.j$
$\Rightarrow oq.j = oq.k \ \wedge \ k$ **in** $[17 \ldots 30]$,

Lq3: j **in** $[21 \ldots 32] \ \wedge \ k$ **in** $[14 \ldots 17] \ \wedge \ oq.j = oq.k$
$\Rightarrow k \in liSy.j \ \vee \ \{j\} \cup est.j \subseteq liDo.k \cup est.k.$

Nq0: j **at** $23 \ \wedge \ prio.j \ \wedge \ k = th.j \ \wedge \ k$ **in** $[18 \ldots 32] \ \wedge \ oq.j = oq.k$
$\Rightarrow tkk.j < tkk.k$,
Nq1: j **at** $24 \ \wedge \ k = th.j \ \wedge \ k$ **in** $[18 \ldots] \ \wedge \ oq.j = oq.k \ \Rightarrow \ tkk.j < tkk.k$,
Nq2: j **at** $25 \ \wedge \ k = th.j \ \wedge \ k$ **in** $[18 \ldots 32] \ \wedge \ oq.j = oq.k \ \Rightarrow \ tkk.j < tkk.k$,
Nq3: j **in** $[22 \ldots 33] \ \wedge \ k$ **in** $[18 \ldots 33] \ \wedge \ oq.j = oq.k$
$\Rightarrow j = k \ \vee \ k \in liPri.j \ \vee \ tkk.j < tkk.k.$

Pq0: j **in** $[14 \ldots 18] \ \wedge \ k \in \texttt{predec}[j] \ \wedge \ oq.j = oq.k$
$\Rightarrow \ \{k\} \cup est.k \subseteq est.j \cup liDo.j$,
Pq1: $k \in \texttt{predec}[j] \ \Rightarrow \ k$ **in** $[21 \ldots 32]$,
Pq2: j **in** $[19 \ldots] \ \wedge \ k \in \texttt{predec}[j] \ \wedge \ oq.j = oq.k \ \Rightarrow \ tkk.k < tkk.j$,
Pq3: j **in** $[22 \ldots] \ \Rightarrow \ \texttt{predec}[j] \subseteq liPri.j$,
Pq4: $j \notin \texttt{predec}[j]$,
Pq5: j **in** $[13 \ldots 33] \ \wedge \ k \in \texttt{predec}[j] \ \Rightarrow \ oq.j = \texttt{nwq} \ \vee \ oq.j = oq.k$,
Pq6: j **at** $23 \ \wedge \ k \in \texttt{predec}[j] \ \wedge \ oq.j = oq.k \ \wedge \ prio.j \ \Rightarrow \ th.j \neq k$,
Pq7: j **at** $24 \ \wedge \ k \in \texttt{predec}[j] \ \wedge \ oq.j = oq.k \ \Rightarrow \ th.j \neq k$,
Pq8: j **at** $25 \ \wedge \ k \in \texttt{predec}[j] \ \Rightarrow \ th.j \neq k$,
Qq0: j **in** $[13 \ldots 30] \ \Rightarrow \ oq.j = \texttt{nwq} \ \vee \ oq.j = \texttt{owq} \ \vee \ j \in \texttt{admit}.$

Some derived invariants:

qFCFS: j **in** $[26 \ldots] \ \Rightarrow \ \texttt{predec}[j] = \emptyset$,
Dq0: $\texttt{wq} = \texttt{nwq} \ \vee \ \texttt{wq} = \texttt{owq}$,
Dq1: j **in** $[13 \ldots 17] \ \Rightarrow \ oq.j = \texttt{nwq} \ \vee \ oq.j = \texttt{owq}$,
Dq2: j **in** $[26 \ldots 27] \ \Rightarrow \ \texttt{wq} = \texttt{nwq} = \texttt{owq}$,
Dq3: j **in** $[31 \ldots 33] \ \Rightarrow \ \texttt{wq} = \texttt{nwq} = \texttt{owq}$,
Dq4: j **at** $27 \ \wedge \ k$ **in** $[13 \ldots 33] \ \Rightarrow \ oq.k = \texttt{nwq}$,
Dq5: j **in** $[13 \ldots 17] \ \wedge \ k$ **in** $[30 \ldots 33] \ \wedge \ liSw = \emptyset \ \Rightarrow \ oqj \neq oq.k$,
Dq6: j **in** $[20 \ldots 33] \ \wedge \ oq.j = \texttt{nwq} \ \Rightarrow \ \texttt{wq} = \texttt{nwq}$,
Kq1d: j **at** $22 \ \wedge \ k$ **in** $[26 \ldots 33] \ \Rightarrow \ k \in liPri.j \ \vee \ oq.j = oq.k$,
Lq2d: j **in** $[21 \ldots 33] \ \wedge \ k$ **in** $[13 \ldots 17] \ \wedge \ oq.j = oq.k \ \wedge \ h \in est.j$
$\Rightarrow \ oq.h = oq.j \ \wedge \ h$ **in** $[18 \ldots 30]$,
Lq3d: j **in** $[22 \ldots 30] \ \wedge \ k$ **at** $17 \ \wedge \ oq.j = oq.k \ \Rightarrow \ tkk.j < tkk.k$,
Pq0d: j **at** $18 \ \wedge \ k \in \texttt{predec}[j] \ \wedge \ oq.j = oq.k \ \Rightarrow \ tkk.k < tkk.j$,
Pq1d: j **at** $13 \ \wedge \ k \in \texttt{predec}[j] \ \wedge \ oq.j = oq.k \ \Rightarrow \ j \notin est.r.$

References

1. Abadi, M., Lamport, L.: The existence of refinement mappings. Theor. Comput. Sci. **82**, 253–284 (1991)
2. Abraham, U.: Bakery algorithms. In: Proceedings of the Concurrency, Specification, and Programming Workshop, pp. 7–40 (1993)
3. Anderson, J.H.: A fine-grained solution to the mutual exclusion problem. Acta Inf. **30**(3), 249–265 (1993)
4. Anderson, J.H., Kim, Y.J., Herman, T.: Shared-memory mutual exclusion: major research trends since 1986. Distrib. Comput. **16**, 75–110 (2003)
5. Aravind, A.: An arbitration algorithm for multiport memory systems. IEICE Electron. Express **2**, 488–494 (2005)
6. Aravind, A.A., Hesselink, W.H.: A queue based mutual exclusion algorithm. Acta Inf. **46**, 73–86 (2009)
7. Dijkstra, E.W.: Solution of a problem in concurrent programming control. Commun. ACM **8**, 569 (1965)

8. Dijkstra, E.W.: Co-operating sequential processes. In: Genuys, F. (ed.) Programming Languages, London, etc, pp. 43–112. NATO Advanced Study Institute, Academic Press, London (1968)
9. Haldar, S., Subramanian, P.S.: Space-optimum conflict-free construction of 1-writer 1-reader multivalued atomic variable. In: Proceedings of the 8th International Workshop on Distributed Algorithms, volume 857 of LNCS, pp. 116–129. Springer, Berlin (1994)
10. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann Publishers, Los Altos (2008)
11. Hesselink, W.H.: A challenge for atomicity verification. Sci. Comput. Program. **71**, 57–72 (2008)
12. Hesselink, W.H.: PVS proof scripts of "nonatomic bakery algorithm with bounded tokens". Available at http://www.cs.rug.nl/~wim/mechver/bnonatomicMX/index.html (2009)
13. Holzmann, G.J.: The SPIN Model Checker, Primer and Reference Manual. Addison-Wesley, Reading (2004)
14. Inoue, T., Hironaka, T., Sasaki, T., Fukae, S., Koide, T., Mattausch, H.J.: Evaluation of bank-based multiport memory architecture with blocking network. Electron. Commun. Jpn **89**, 498–510 (2006)
15. Israeli, A., Li, M.: Bounded time-stamps. Distrib. Comput. **6**, 205–209 (1993)
16. Jayanti, P., Tan, K., Friedland, G., Katz, A.: Bounding Lamport's Bakery algorithm. In: Proceedings of the SOFSEM, volume 2234 of LNCS, pp. 261–270 (2001)
17. Katseff, H.P.: A new solution to the critical section problem. In: Tenth Annual ACM Symposium on Theory of Computing, pp. 86–88 (1978)
18. Kessels, J.L.W.: Arbitration without common modifiable variables. Acta Inf. **17**, 135–141 (1982)
19. Kim, Y.J., Anderson, J.H.: Nonatomic mutual exclusion with local spinning. Distrib. Comput. **19**, 19–61 (2006)
20. Lamport, L.: A new solution of Dijkstra's concurrent programming problem. Commun. ACM **17**, 453–455 (1974)
21. Lamport, L.: The mutual exclusion problem—part I: a theory of interprocess communication, part II: statement and solutions. J. ACM **33**, 313–348 (1986)
22. Lamport, L.: On interprocess communication. Parts I and II. Distrib. Comput. **1**, 77–101 (1986)
23. Lamport, L.: The mutual exclusion problem has been solved. Commun. ACM **34**(1), 110,119 (1991)
24. Lamport, L.: The temporal logic of actions. ACM Trans. Program. Lang. Syst. **16**, 872–923 (1994)
25. Lycklama, E.A., Hadzilacos, V.: A first-come-first-served mutual-exclusion algorithm with small communication variables. ACM Trans. Program. Lang. Syst. **13**, 558–576 (1991)
26. Owicki, S., Gries, D.: An axiomatic proof technique for parallel programs. Acta Inf. **6**, 319–340 (1976)
27. Owre, S., Shankar, N., Rushby, J.M., Stringer-Calvert, D.W.J.: PVS Version 2.4, System Guide, Prover Guide, PVS Language Reference. http://pvs.csl.sri.com (2001)
28. Peterson, G.L.: Myths about the mutual exclusion problem. Inf. Process. Lett. **12**, 115–116 (1981)
29. Peterson, G.L.: A new solution to Lamport's concurrent programming problem using small shared variables. ACM Trans. Program. Lang. Syst. **5**(1), 56–65 (1983)
30. Raynal, M.: Algorithms for Mutual Exclusion. MIT Press, Cambridge (1986)
31. Shiue, W.-T., Chakrabarti, C.: Multi-module multi-port memory design for low power embedded systems. Design Autom. Embed. Syst. **9**, 235–261 (2004)
32. Takamura, M., Igarashi, Y.: Simple mutual exclusion algorithms based on bounded tickets on the asynchronous shared memory model. In: Proceedings of the COCOON, volume 2387 of LNCS, pp. 259–268 (2002)
33. Taubenfeld, G.: The black-white bakery algorithm and related bounded-space, adaptive, local-spinning and FIFO algorithms. In: Proceedings of the DISC, volume 3274 of LNCS, pp. 56–70 (2004)
34. Taubenfeld, G.: Synchronization Algorithms and Concurrent Programming. Pearson Education/Prentice Hall, Englewood Cliffs (2006)
35. Vijayaraghavan, S.: A variant of the bakery algorithm with bounded values as a solution to Abraham's concurrent programming problem. In: Proceedings of Design, Analysis and Simulation of Distributed Systems (2003)
36. Vitányi, P.M.B.: Registers. In: Encyclopedia of Algorithms, pp. 761–764. Springer, Berlin (2008)
37. Vitányi, P.M.B., Awerbuch, B.: Atomic shared register access by asynchronous hardware. In: 27th Annual Symposium on Foundations of Computer Science, pp. 233–243. IEEE, Los Alamitos, California, 1986. Corrigendum in 28th Annual Symposium on Foundations of Computer Science, p. 487. Los Angeles (1987)
38. Woo, T.-K.: A note on Lamport's mutual exclusion algorithm. SIGOPS Oper. Syst. Rev. **24**(4), 78–81 (1990)
39. Zuo, W., Qi, Z., Jiaxing, L.: An intelligent multi-port memory. In: Proceedings of the IEEE International Symposium on Information Technology Application Workshops, pp. 251–254 (2008)